Merely Useful: Novice R

Madeleine Bonsma-Fisher, Kate Hertweck, Damien Irving, Luke Johnston, Christina Koch

Contents

List of Tables

List of Figures

1 Overview

It's still magic even if you know how it's done.
— Terry Pratchett

FIXME: general introduction

- Be able to do the steps in Yenni et al. (2019).

1.1 WHY ISN'T ALL OF THIS NORMAL ALREADY?

Nobody argues that research should be irreproducible or unsustainable, but "not against it" and actively supporting it are very different things. Academia doesn't yet know how to reward people for writing useful software, so while you may be thanked, the extra effort you put in may not translate into job security or decent pay.

And some people still argue against openness, Being open is a big step toward a (non-academic) career path, which is where approximately 80% of PhDs go, and for those staying in academia, open work is cited more often than closed (FIXME: citation). However, some people still worry that if they make their data and code generally available, someone else will use it and publish a result they have come up with themselves. This is almost unheard of in practice, but that doesn't stop people using it as a boogeyman.

Other people are afraid of looking foolish or incompetent by sharing code that might contain bugs. This isn't just impostor syndrome: members of marginalized groups are frequently judged more harshly than others (FIXME: citation).

1.2 ACKNOWLEDGMENTS

This book owes its existence to the hundreds of researchers we met through Software Carpentry and Data Carpentry. We are also grateful to Insight Data Science for sponsoring the early stages of this work, to everyone who has contributed, and to:

- *Practical Computing for Biologists* Haddock and Dunn (2010)
- *Effective Computation in Physics* Scopatz and Huff (2015)
- "A Quick Guide to Organizing Computational Biology Projects" Noble (2009)
- "Ten Simple Rules for Making Research Software More Robust" Taschuk and Wilson (2017)
- "Best Practices for Scientific Computing" Wilson et al. (2014)
- "Good Enough Practices in Scientific Computing" Wilson et al. (2017)

2 Novice Goals

This outline describes the questions that the novice courses on R and Python will answer. The advanced course can then assume that learners have hands-on experience with these topics but nothing more.

2.1 PERSONAS

2.1.1 ANYA

Anya is a professor of neuropsychology who is responsible for teaching her department's introduction to statistics to 1100 first-year students every year. (Students complain that the Stats department's introductory course is too theoretical and requires more programming knowledge than they have.) When she finds time for it, her research focuses on color perception in infants.

Over the past nine years, Anya has designed and run a dozen experiments on 50-100 infant subjects each and analyzed the results using SPSS and more recently R (which she taught herself during a sabbatical). She has never taken a programming course, and suffers from impostor syndrome when talking to colleagues who are using things like GitHub and R Markdown.

Anya would like to figure out how to use R to teach her intro stats course, which currently uses a mixture of Excel and SPSS. She would like to learn more about time series analysis to support her research, and about tools like Git and R Markdown.

This guide has modular lessons and exercises that she can adapt to use in her course, and suggestions for how to make learning interactive with a large class size. She also finds helpful instructions for applying time series analysis to data using R.

2.1.2 EXTON

Exton taught business at a community college before joining a friend's startup, and now does community management for a company that

builds healthcare software. He still teaches Marketing 101 every year to help people with backgrounds like his.

Exton uses Excel to keep track of who is registered for webinars, workshops, and training sessions. Some of these spreadsheets are created from CSV files produced by a web-scraping script a summer intern wrote for him a couple of years ago. Exton doesn't think of himself as a programmer, but spends hours creating complicated lookup tables in multi-sheet spreadsheets to help him figure out how many webinar attendees turn into community contributors, who answers forum posts most frequently, and so on.

Exton knows there are better ways to do what he's doing, but feels overwhelmed by the flood of blog posts, tweets, and "helpful" recommendations he receives from members of the company's engineering team. He wants someone to tell him where he should start and how long it will take whatever he learns to pay off.

Exton finds 'Merely Useful' after some Googling, and sees an example of data analysis with spreadsheet data that looks really similar to what he's trying to do. He carefully works through that particular example, then goes back and works through some of the earlier material in the book. He can tell that it won't take long to get this to work with his data.

2.1.3 IRWIN

Irwin, 18, is five months into an undergraduate degree in urban planning. He's read lots of gushing articles in *Wired* about data science, and was excited by the prospect of learning how to do it, but dropped his CS 101 course after six weeks because nothing made sense. (His university's computer science department uses Haskell as an introductory programming language…) He is doing better in Anya's course (which he is taking as an elective) but still spends most of his time copying, pasting, and swearing.

Irwin did well in his high school math classes, and built himself a home page with HTML and CSS in a weekend workshop in grade 11. He knows how to do simple calculations in Excel, has accounts on nine different social media sites, and attends all of his morning classes online.

Anya mentions this guide in one of her classes, and Irwin downloads the PDF to read on the bus. He loves the examples that use urban data, and right away he has tons of ideas about where to get more cool data

to analyze. His urban data science blog is already taking shape in his head.

2.1.4 CAMILLA

Camilla recently started a job as an assistant professor. Her department (Medieval Studies) is trying to develop a digital humanities data-science-heavy undergraduate program, and the undergraduate chair thinks that Camilla has the most programming experience in the department and has asked her to develop an introduction to programming course for humanities students.

Camilla has dabbled in natural language processing and has learned Python over the course of her previous work, but she has no experience teaching progamming and she's not sure what the best way is to teach beginners. She doesn't want to start from scratch to create a course out of nothing. She also isn't sure which programming language the new program should focus on.

She finds 'Merely Useful' and feels relieved: she can pretty much use the book as-is for her course. She looks up the examples of text and image analysis and compares how both R and Python approach those kinds of data to help her make a decision about which language to teach.

2.1.5 JORDAN

Jordan is a third-year undergraduate student in ecology. Two months ago she started working part-time for a professor in her department, and she's beginning to collect and analyze data from her own experiments with fruit flies. Her professor has asked her to learn R to do her analysis and suggested that she sign up for the introduction to quantitative data analysis in R course that the ecology department offers. The course is just starting, and it uses 'Merely Useful' as the textbook.

Jordan can't wait to apply her new programming knowledge to her data, so she starts reading ahead and trying to use her own data in some of the book's examples. As she works through examples, she realizes that she'll need to change a few things about how she records her data in spreadsheets so that it will be easier to analyze in R.

2.2 GETTING STARTED

- What are the different ways I can interact with software?

  - console
  - scripts
- How can I find and view help?
  - In the IDE
  - Stack Overflow
- How can I inspect data while I'm working on it?
  - table viewers

2.3 DATA MANIPULATION

- How can I read tabular data into a program?
  - what CSV is, where it comes from, and why people use it
  - reading files
- How can I select subsets of my data?
  - select
  - filter
  - arrange
  - Boolean conditions
- How can I calculate new values?
  - mutate
  - ifelse
- How can I tell what's gone wrong in my programs?
  - reading error messages
  - the difference between syntax and runtime errors
- How can I operate on subsets of my data?
  - group
  - summarize
  - split-apply-combine
- How can I work with two or more datasets?
  - join
- How can I save my results?
  - writing files
- What *isn't* included?
  - anything other than reasonably tidy tabular data
  - map
  - loops and conditionals

2.4 PLOTTING

- Why plot?
  - summary statistics can mislead
  - Anscombe's Quartet and the DataSaurus dozen
- What are the core elements of every plot?
  - data
  - geometric objects
  - aesthetic mapping
- How can I create different kinds of plots?
  - scatter plot
  - line plot
  - histogram
  - bar plot
  - which to use when
- How can I plot multiple datasets at once?
  - grouping
  - faceting
- How can I make misleading plots?
  - showing a single central tendency data point instead of the individual observations
  - saturated plots instead of for example violins or 2D histograms
  - picking unreasonable axes limits to intentionally misrepresent the underlying data
  - not using perceptually uniform colormaps to indicate quantities
  - not thinking about color blindness
- What *isn't* included?
  - outliers
  - interactive plots
  - maps
  - 3D visualization

2.5 DEVELOPMENT

- How can I make my own functions?
  - declaring functions
  - declaring parameters
  - default values

    - common conventions
- How can I make my programs tell me that something has gone wrong?
    - validation (did we build the right thing) vs. verification (did we build the thing correctly)
    - assertions for sanity checks
- How can I ask for help?
    - creating a reproducible example (reprex)
- How do I install software?
    - what *is* a package?
    - package manager
- What *isn't* included?
    - code browsers, multiple cursors, and other fancy IDE tricks
    - virtual environments
    - debuggers

2.6 DATA ANALYSIS

- How can I represent and manage missing values?
    - NA
- How can I get a feel for my data?
    - summary statistics
- How can I create a simple model of my data?
    - formulas
    - linear regression
        - adding a best fit straight line on a scatterplot
        - understanding what the error bands on a "best fit" straight line mean
    - k-means cluster analysis
    - frame these as exploratory tools for revealing structure in the data, rather than modelling or inferential tools
- How can I put people at risk?
    - algorithmic bias
    - de-anonymization
- What *isn't* included?
    - statistical tests
    - multiple linear regression
    - anything with "machine learning" in its name

2.7 VERSION CONTROL

- What is a version control system?
  - a smarter kind of backup
- What goes where and why?
  - local vs. remote storage (physically)
  - local vs. remote storage (ethical/privacy issues)
- How do I track my work locally?
  - diff
  - add
  - commit
  - log
- How do I view or recover an old version of a file?
  - diff
  - checkout
- How do I save work remotely?
  - push and pull
- How do I manage conflicts?
  - merge
- What *isn't* included?
  - forking
  - branching
  - pull requests
  - git reflow –substantive –single-afferent-cycle –ia-ia-rebase-fhtagn …

2.8 PUBLISHING

- How do static websites work?
  - URLs
  - servers
  - request/response cycle
  - pages
- How do I create a simple HTML page?
  - head/body
  - basic elements
  - images
  - links
  - relative vs. absolute paths

- How can I create a simple website?
  - GitHub pages
- How can I give pages a standard appearance?
  - layouts
- How can I avoid writing all those tags?
  - Markdown
- How can I share values between pages?
  - flat per-site and per-page configuration
  - variable expansion
- What *isn't* included?
  - templating
  - filters
  - inclusions

2.9 REPRODUCIBILITY

- How can I make programs easy to read?
  - coding style
  - linters
  - documentation
- How can I make programs easy to re-use?
  - Taschuk's Rules
- How can I combine explanations, code, and results?
  - notebooks
- Where does stuff actually live on my computer?
  - directory structure on Windows and Unix
  - absolute vs. relative paths
  - significance of the working directory
  - data on disk vs. data in memory
- How should I organize my projects?
  - Noble's Rules
  - RStudio projects
- How should I keep track of my data?
  - simple manifests
- What *isn't* included?
  - build tools (Make and its kin)
  - continuous integration
  - documentation generators

2.10 COLLABORATION

- What kinds of licenses are there?
  - open vs. closed
  - copyright
- Who gets to decide what license to use?
  - it depends…
- What license should I use for my publications?
  - CC-something
- What license should I use for my software?
  - MIT/BSD vs. GPL
- What license should I use for my data?
  - CC0
- How should I identify myself and my work?
  - DOIs
  - ORCIDs
- How do I credit someone else's code?
  - citing packages, citing something from GitHub, giving credit for someone's answer on StackOverflow…
- What's the difference between open and welcoming?
  - evidence for systematic exclusion
  - mechanics of exclusion
- How can I help create a level playing field?
  - what's wrong with deficit models
  - allyship, advocacy, and sponsorship
  - Code of Conduct (remove negatives)
  - curb cuts (adding positives for some people helps everyone else too)
- What *isn't* included?
  - how to run a meeting
  - community management
  - mental health
  - assessment of this course

3 Introduction

FIXME: general introduction.

3.1 WHO ARE THESE LESSONS FOR?

3.1.1 EXTON EXCEL

1. Exton taught business at a community college for several years, and now does community management for an event management company. He still teaches Marketing 101 every year to help people with backgrounds like his.
2. Exton uses Excel to keep track of who is registered for webinars, workshops, and training sessions. He doesn't think of himself as a programmer, but spends hours creating complicated lookup tables to figure out how many webinar attendees turn into community contributors, who answers forum posts most frequently, and so on.
3. Exton knows there are better ways to do what he's doing, but feels overwhelmed by the blog posts, tweets, and "helpful" recommendations from the company's engineering team.
4. Exton is a single parent; the one evening a week he spends teaching is the only out-of-work time he can take away from family responsibilities.

3.1.2 NINA NEWBIE

1. Nina is 18 years old and in the first year of an undergraduate degree in urban planning. She has read lots of gushing articles about data science, and was excited by the prospect of learning how to do it, but dropped her CS 101 course after six weeks because nothing made sense. She is doing better in her intro to statistics (which uses a little bit of R), but still spends most of her time copying, pasting, and swearing.
2. Nina did well in her high school math classes, and built herself a home page with HTML and CSS in a weekend workshop in grade 11. She has accounts on nine different social media site, and attends all of her morning classes online.

3. Nina wants self-paced tutorials with practice exercises, plus forums where she can ask for help.
4. After a few bruising conversations with CS majors, Nina is reluctant to reveal how little she knows about programming: she would rather get a low grade and blame it on partying than let her classmates see that she is floundering.

3.2 WHAT WILL THESE LESSONS TEACH YOU?

During this course, you will learn how to:

- Write programs in R that read data, clean it up, perform simple statistical analyses on it.
- Build visualizations to help you understand your data and communicate your findings.
- Find and install R packages to help you do these things.
- Create software that other people can understand and re-run.
- Track your work in version control using Git and GitHub.
- Publish your results on a web site using R Markdown and GitHub Pages.
- Make your data, software, and reports citable using ORCIDs and DOIs, and cite the work of others.
- Select open source and Creative Commons licenses that allow you and others to share data, software, and reports.
- Be an active participant in open, inclusive projects.

3.2.1 WHAT DO YOU NEED TO HAVE AND KNOW TO START?

This course assumes that you:

- Have a laptop that you can install software on, or access to a web-based programming system like rstudio.cloud.
- Know what mean and variance are.
- Are willing to invest about 30 hours in reading or lectures and another 100 hours doing practice exercises.

We have tried to make these lessons accessible to people with visual or motor challenges, but recognize that some parts (particularly data visualization) may still be difficult. We welcome suggestions for improvements.

3.3 WHAT EXAMPLES WILL WE USE?

FIXME: introduce running examples

3.4 WHAT'S THE BIG PICTURE? (#R-INTRO-BIGPICTURE}

We now swim in a sea of data and generate more each day. That data can help us understand the world, but it can also be used to manipulate us and invade our privacy. Learning how to analyze data will help you do the former and guard against the latter.

This course is therefore about *people*, *programs*, and *data*. Data can live in three places (FIXME: diagram)

1. In the computer's memory. It has to be here for the computer to use it, but when a program stops running or the computer is shut down, the contents of memory evaporate.
2. On the computer's hard drive. This is much larger than memory—terabytes instead of gigabytes—and its contents are organized into *files* and *directories* (also called *folders*). What's on the hard drive stays there even when programs aren't running or the computer is switched off. A program must *read* data from files into memory to work with it, and *write* data to files to save it permanently.
3. On some other computer on the network. "The cloud" and "the web" are just other people's computers; if we want to use data that's on the other side of a URL, we need to download it (i.e., copy it to our hard drive) or read it directly into memory (which is called streaming).

Programs also live in three places (FIXME: enhanced diagram)

1. In the computer's memory. A program has to be in memory for the computer to run it.
2. On the computer's hard drive. A program is a file like any other. Instead of containing text, pixels, or CO2 measurements, it contains instructions. In order for the computer to run it, those instructions have to be copied into memory. (This is part of what your computer is doing when it launches an application.) Your program typically uses other pieces of software called packages or libraries that provide common operations like searching

in text, changing the colors of pixels, or calculating averages. When your program is loaded into memory, your computer also loads those packages.
3. On some other computer on the network. R and Python both have catalogs of packages that people have written and shared. In order to use one of these, you must install it on your computer by copying its files from the catalog site to your hard drive. There's usually more to this than simply copying one file, so R and Python both come with tools to help you find, install, and manage packages.

Finally, we come to people (FIXME: enhanced diagram)

1. This course starts by teaching you how to write programs that run on your computer, analyze data that is on your computer or on the web, and use packages written by other people.
2. It will also teach you how to write programs that your collaborators can understand and run. (One of those collaborators is your future self, who will be grateful three months from now that you did the right thing today.) These skills will make you a productive member of a small team, and this course will also explain how to make such teams open and inclusive.
3. The third group of people includes collaborators or reviewers who want to reproduce your work. This course will show you how to publish your results on the web and how to get credit for your work and give it to others.

4 Getting Started with R

4.1 QUESTIONS

- How do I run R code?
- What are some basic actions I can do with R?
- How do I get help?

As stated in the introduction, our overall goal is to work with people, programs and data. In this section, we will focus on programs and data as we learn how to run R code, as well as how data is stored and accessed on a computer.

4.2 INTRODUCTION TO RSTUDIO

Throughout this book, we'll be writing programs (or, in verb form, *programming*) in order to accomplish our goals of working with data on the computer. Programming is one way to make a computer do something for us. Instead of clicking, we'll mostly be typing; instead of doing what someone else has pre-defined, we'll have a lot of flexibility to do what we want.

Just like using a web browser to access websites, and a program like MS Word to write documents, it's helpful to have a program on your computer that is designed to make it easy to write and run code. This kind of program is called an "IDE" or Integrated Development Environment.

The one we'll be using in this book for writing R code is called RStudio. RStudio (like many IDEs) has many panes (or panels or boxes), each of which has a different purpose.

FIXME: Screenshot or schematic.

We'll take it slowly and introduce the purpose of each pane one at a time. We'll start with the pane occupying the lefthand side of your screen.

4.3 RUNNING CODE IN THE CONSOLE

Now that we have our space for writing and running code (our **environment**) open, it's time to actually run some code.

The first time you open a new installation of RStudio, the pane occupying the entire lefthand side of the screen is called the console. The console is a program that is constantly ready and waiting to accept and run code. The console you see in RStudio is expecting to see R code. Here's an example you can type or copy in to see how the console works:

```
print("It was the best of times, it was the worst of times.")
```

```
## [1] "It was the best of times, it was the worst of times."
```

A function is a set of code that can repeatedly perform a specific task. As you might guess from the name, the R `print()` function takes in text (indicated by the quotes) and then prints the text back out to the console. R has many built-in functions, each of which accomplishes different things:

```
round(3.1415)
```

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

```
Sys.Date()
```

```
## [1] "2019-11-07"
```

```
length("hippo")
```

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

Part of learning to program is learning some of the base R functions and what they do; in upcoming chapters, we'll focus on functions that allow you to read in and manipulate data. At some point it may be helpful to write your *own* functions, which we'll also cover in another chapter.

4.4 RUNNING CODE VIA SCRIPTS

Can you imagine a situation where continuing to type code into the console could become tedious or challenging?

Here are some examples:

- Running the same code many times (you *can* use the up-arrow to see previous commands - try it! - but if you run many commands, you might end up scrolling a long time.
- Organizing long sections of code in a meaningful way, and working on subsections separately.
- Saving the code as a text file that can be stored on the computer's hard drive and shared with collaborators.

Clearly, we need an approach that will allow us to write and run code while keeping a record of our work and allowing us to run (and re-run) parts.

The answer to these challenges is to add our R code to a text file called a script. You can create a script in RStudio by using the "File" menu option to select "New file" and then "R Script." Your R script (a blank text file) should appear on the left side of the screen above the console. Try copying in some of the commands we've already run:

```
round(3.1415)
```

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

```
Sys.Date()
```

```
## [1] "2019-11-07"
```

FIXME: Screenshot or animated gif.

Now these commands are written in the script, but nothing has happened yet. In order to run the commands from the script (just like we did in the console), we can choose from a variety of run commands. RStudio includes a "Run" button in the script pane. Clicking this button runs the current line of code. You should see both the code and output appear in the console pane on the bottom left. Your cursor will also move to the next line of code. Alternatively, you can use a keyboard shortcut to run code. Hovering your mouse over the run button will show you the shortcut for running code on your computer; this shortcut is typically `Ctrl+Enter`. You can also try selecting multiple lines of code with your cursor to execute multiple lines simultaneously.

Sometimes, when you want to experiment or check something, it can make sense to write and run R code in the console. However, most of

the time, you'll want to write and run your code from a script. Using scripts has the benefit of saving your work while also being able to run the code just like in the console. You and your collaborators can then also easily share and run the code outside of the IDE.

We have now told the computer what to do by using R code, and we have run that code in two different ways within our "workbench" - the RSudio IDE. Let's see what else we can use in this environment to help us.

4.5 ASSIGNING AND RECALLING OBJECTS

Besides running functions that *do* something (as above), we'll also want to use R to keep track of information that we're using throughout our analysis. We save that information by creating an object) using a name we select, special set of symbols known as the assignment operator (`<-`), and then the information to save. For example:

```
message <- "It is a far, far better thing that I do, than I have ever done"
name_length <- length("Sydney Carton")
```

On the top right pane, there's a tab that says "Environment". If you click on that, you'll see the objects you just created listed under "Values." Objects can represent data of various *types* - character (collection of letters and numbers encased in quotation marks), numeric (numbers, including decimals) and more.

Referencing the name of an object by itself will print the data you assigned to the console. You can also use objects as input to a function:

```
message
```

```
## [1] "It is a far, far better thing that I do, than I have ever done"
```

```
class(message)
```

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

These examples are objects representing small, fairly simple data. Objects can also be much more complex, representing large datasets. Hopefully you can imagine how assigning objects can allow you to work more

easily with complex data, which you'll have a chance to do in the next section.

4.6 VIEWING INSTALLED PACKAGES

Functions in R are grouped together into collections of related code called packages. There is a tab in the lower right pane in RStudio called "Packages" that lists all of the packages installed in your version of R. Scrolling down the list, you'll note that some packages have the box on the left checked. This means the package has been loaded and is available for use.

In addition to the built-in packages that come with every installation of R, we have the ability to use packages written by other programmers. One of the most common places to find such code is the Comprehensive R Archive Network, or CRAN. RStudio allows a straightforward way to download and install packages through CRAN. If you click on the "Install" button in the Packages tab, a window will appear allowing you to install from "Repository (CRAN)." In the space below "Packages," type "tidyverse." Making sure the box next to "Install dependencies" is checked, click "Install" to download and install this collection of code (which we'll be using in later sections).

If you'd prefer to use code to install packages, rather than the RStudio pane, you could execute the following line of code in your console:

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

This accomplishes the same task as searching and installing using the Packages pane. We don't recommend including this code in your script, since you won't need to install the package every time you run the script.

Once a package is installed, you'll need to ensure it is loaded and all functions within it are recognized by R for use. You can load a package by locating it among installed packages in the Packages pane, and then checking the box to the left of its name. Alternatively, you can perform the same task using code:

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

It is appropriate to include this code in your script, so any functions from loaded packages are recognized by R and run as expected.

4.7 GETTING HELP IN RSTUDIO

There are a number of additional features in RStudio that may be of use, but we'll discuss them in later sections as the need arises. The remaining tab that will be useful to us now is "Help," located in the lower right pane.

You can search for more information on functions in R by typing the name of the function, such as "round", into the search box in the Help pane. The text that appears in the window below can sometimes be extensive. You can search within this particular help page using the second search box that reads "Find in Topic." In this case, type "decimal" to find information about how the function determines the number of decimal points to print in the output.

Luckily, R help documentation tends to be formatted very consistently. At the very top, you'll see the name of the function ("Round", which in this case represents a collection of related functions) followed by the name of its package inside curly brackets ("{base}", this is important because sometimes different packages have functions with the same names!). Below that, a short title indicates the purpose of the function. "Description" is a more extensive explanation that should help you figure out if the function is appropriate for your use. "Usage" provides information on how the function should be applied through code, and "Arguments" is like a legend for the code described in Usage. The last subheadings, "Details" through "References" and "Examples", should be self-explanatory.

When you're writing code in your script, you can also access help documentation by prefacing the name of the function with a question mark. Similar to installing packages, searching help documentation is useful when you're working on writing code, but isn't particularly helpful to include in your script, so try typing the following code into the console:

```
?sum
```

After executing that code, you should see the help page for that function appear in the lower right pane. This search method works on functions that are currently loaded. If you would like to perform a global search (e.g., search among all R packages installed on your computer), you can use two question marks instead (remember to enter this in the console):

```
??filter
```

RStudio also provides helpful pop-up windows that attempt to predict the function name you're typing. You may see windows that appear as you're coding, which include short versions of the help documentation.

4.8 GETTING HELP ONLINE

Sometimes you may not be able to find the answers you seek in the documentation available through RStudio. This often happens when you need to do something completely new and you don't know where to start. In that scenario, there are multiple internet resources that are helpful:

- blogs
- Twitter
- tutorials
- question pages on Stack Overflow, an open forum that allows community members to ask and answer questions

It can be challenging to find the right words to search when you're troubleshooting something new. Don't be discouraged, and think about different ways you can apply the terminology you're learning to make your search terms clearer and more specific.

Even when you find information that seems useful, these resources can be overwhelming and confusing. Some things to look for in a Stack Overflow discussion include:

- Is the question clearly stated and is there example code? If you're debugging, does the example code look like yours?
- Answers have upvotes and downvotes. Is there one clear answer that has a lot of upvotes in response to a question?
- How complex is the answer? While some questions will necessarily have a complicated answer, for many common programming tasks, there should be a solution that only takes a few lines of code.
- Do you recognize any terms in the solution? If you don't, are there other terms for which you could search?

Conventions:
One tool in using help pages (and in reading the rest of the book) will be understanding the conventions that writers use when describing many of the terms introduced in this chapter. For example

`min()`

indicates that something is a function, with parentheses to indicate that it is an action with (potentially) input and output.
Some other conventions used in this book are: * `folder_name/` for a folder * `variable_name` for variables * `column_name` for columns

4.9 EXERCISES

1. Run each of the functions above. Can you explain what each function expects as input, and what kind of output it produces?
2. What would be the pros and cons of using the console versus a script in each situation?
   - Writing a data analysis with multiple steps.
   - Opening a new data set and exploring its dimensions.
   - Checking the value of a variable.
3. After googling "import csv file r", the two following pages appear in the search results: https://stackoverflow.com/questions/3391880/how-to-get-a-csv-file-into-r http://www.sthda.com/english/wiki/reading-data-from-txt-csv-files-r-base-functions
   - What is useful about these two pages?
   - Do you understand all of the code in the second page?
   - The second page provides many possible options for code to import a csv file. Where would you start in attempting to perform this task?
4. Try reading a csv file:

   ```
   data <- read_csv("measurements.csv")
   ```

   Which pane in RStudio will show our new `data` object? What happens when you click on it?

4.10 KEY POINTS

- Spyder is an Integrated Development Environment (IDE) for writing Python code.

- You can run Python via a script or console; for most scenarios, we write and execute code from a script.
- IDEs like Spyder have shortcuts and help pages to facilitate writing code.
- Actions in Python are performed by functions and methods.
- Information in Python is stored as objects that are labeled with variables.

5 Practice

We have covered a lot in the last few lessons, so this one presents some practice exercises to ground what we have learned and introduce a few more commonly-used functions.

5.1 WORKING WITH A SINGLE TIDY TABLE

1. Load the tidyverse collection of package and the `here` package for constructing paths:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

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

```
library(here)
```

```
## here() starts at /home/travis/build/merely-useful/merely-useful.github.io
```

2. Use `here::here` to construct a path to a file and `readr::read_csv` to read that file:

```
path <- here::here("data", "person.csv")
person <- readr::read_csv(path)
```

```
## Parsed with column specification:
## cols(
##   person_id = col_character(),
##   personal_name = col_character(),
##   family_name = col_character()
## )
```

```
person
```

```
## # A tibble: 5 x 3
##   person_id personal_name family_name
##   <chr>     <chr>         <chr>
## 1 dyer      William       Dyer
## 2 pb        Frank         Pabodie
## 3 lake      Anderson      Lake
## 4 roe       Valentina     Roerich
## 5 danforth  Frank         Danforth
```

Read `survey/site.csv`.

3. Count rows and columns using `nrow` and `ncol`:

```
nrow(person)
```

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

```
ncol(person)
```

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

How many rows and columns are in the site data?

4. Format strings using `glue::glue`:

```
print(glue::glue("person has {nrow(person)} rows and {ncol(person)} columns"))
```

```
## person has 5 rows and 3 columns
```

Print a nicely-formatted summary of the number of rows and columns in the site data.

5. Use `colnames` to get the names of columns and `paste` to join strings together:

```
print(glue::glue("person columns are {paste(colnames(person), collapse = ' ')}"))
```

```
## person columns are person_id personal_name family_name
```

Print a nicely-formatted summary of the names of the columns in the site data.

6. Use `dplyr::select` to create a new table with a subset of columns by name:

```
dplyr::select(person, family_name, personal_name)
```

```
## # A tibble: 5 x 2
##   family_name personal_name
##   <chr>       <chr>
## 1 Dyer        William
## 2 Pabodie     Frank
## 3 Lake        Anderson
## 4 Roerich     Valentina
## 5 Danforth    Frank
```

Create a table with just the latitudes and longitudes of sites.

7. Use `dplyr::filter` to create a new table with a subset of rows by values:

```
dplyr::filter(person, family_name < "M")
```

```
## # A tibble: 3 x 3
##   person_id personal_name family_name
##   <chr>     <chr>         <chr>
## 1 dyer      William       Dyer
## 2 lake      Anderson      Lake
## 3 danforth  Frank         Danforth
```

Create a table with only sites south of -48 degrees.

8. Use the pipe operator `%>%` to combine operations:

```
person %>%
  dplyr::select(family_name, personal_name) %>%
  dplyr::filter(family_name < "M")
```

```
## # A tibble: 3 x 2
##   family_name personal_name
##   <chr>       <chr>
## 1 Dyer        William
## 2 Lake        Anderson
## 3 Danforth    Frank
```

Create a table with only the latitudes and longitudes of sites south of -48 degrees.

9. Use `dplyr::mutate` to create a new column with calculated values and `stringr::str_length` to calculate string length:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name))
```

```
## # A tibble: 5 x 4
##   person_id personal_name family_name name_length
##   <chr>     <chr>         <chr>             <int>
## 1 dyer      William       Dyer                  4
## 2 pb        Frank         Pabodie               7
## 3 lake      Anderson      Lake                  4
## 4 roe       Valentina     Roerich               7
## 5 danforth  Frank         Danforth              8
```

Look at the help for the built-in function ***round*** *and then use it to create a table with latitudes and longitudes rounded to integers.*

10. Use `dplyr::arrange` to order rows and (optionally) `dplyr::desc` to impose descending order:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name)) %>%
  dplyr::arrange(dplyr::desc(name_length))
```

```
## # A tibble: 5 x 4
##   person_id personal_name family_name name_length
##   <chr>     <chr>         <chr>             <int>
## 1 danforth  Frank         Danforth              8
## 2 pb        Frank         Pabodie               7
## 3 roe       Valentina     Roerich               7
## 4 dyer      William       Dyer                  4
## 5 lake      Anderson      Lake                  4
```

Create a table sorted by decreasing longitude (i.e., most negative longitude last).

5.2 WORKING WITH GROUPED DATA

1. Read `survey/measurements.csv` and look at the data with `View`:

```
measurements <- readr::read_csv(here::here("data", "measurements.csv"))
```

```
## Parsed with column specification:
## cols(
##   visit_id = col_double(),
##   visitor = col_character(),
##   quantity = col_character(),
##   reading = col_double()
## )
```

```
View(measurements)
```

2. Find rows where `reading` is not NA, save as `cleaned`, and report how many rows were removed:

```
cleaned <- measurements %>%
  dplyr::filter(!is.na(reading))
nrow(measurements) - nrow(cleaned)
```

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

Rewrite the filter expression to select rows where the visitor and quantity are not NA either and report the total number of rows removed.

3. Group measurements by quantity measured and count the number of each (the column is named `n` automatically):

```
cleaned %>%
  dplyr::group_by(quantity) %>%
  dplyr::count()
```

```
## # A tibble: 3 x 2
## # Groups:   quantity [3]
##   quantity     n
##   <chr>    <int>
## 1 rad          8
## 2 sal          7
## 3 temp         3
```

Group by person and quantity measured.

4. Find the minimum, average, and maximum for each quantity:

```
cleaned %>%
  dplyr::group_by(quantity) %>%
  dplyr::summarize(low = min(reading), mid = mean(reading), high = max(reading))
```

```
## # A tibble: 3 x 4
##   quantity    low    mid  high
##   <chr>     <dbl>  <dbl> <dbl>
## 1 rad        1.46   6.56  11.2
## 2 sal        0.05   9.24  41.6
## 3 temp     -21.5  -18.7  -16
```

Look at the range for each combination of person and quantity.

5. Rescale salinity measurements that are greater than 1:

```
cleaned <- cleaned %>%
  dplyr::mutate(reading = ifelse(quantity == 'sal' & reading > 1.0, reading/100, reading))
cleaned
```

```
## # A tibble: 18 x 4
##    visit_id visitor quantity reading
##       <dbl> <chr>   <chr>      <dbl>
##  1      619 dyer    rad         9.82
##  2      619 dyer    sal         0.13
##  3      622 dyer    rad         7.8
##  4      622 dyer    sal         0.09
##  5      734 pb      rad         8.41
##  6      734 lake    sal         0.05
##  7      734 pb      temp      -21.5
##  8      735 pb      rad         7.22
##  9      751 pb      rad         4.35
## 10      751 pb      temp      -18.5
## 11      752 lake    rad         2.19
## 12      752 lake    sal         0.09
## 13      752 lake    temp      -16
## 14      752 roe     sal         0.416
## 15      837 lake    rad         1.46
## 16      837 lake    sal         0.21
## 17      837 roe     sal         0.225
## 18      844 roe     rad        11.2
```

Do the same calculation use `case_when`.

6. Read `visited.csv`, drop the NAs and store in `visits`. Use `anti_join()` to find the measurements in `cleaned` that don't have matches in `visits`:

```
visits <- readr::read_csv(here::here("data", "visited.csv")) %>%
  dplyr::filter(!is.na(visit_date))
```

```
## Parsed with column specification:
## cols(
##   visit_id = col_double(),
##   site_id = col_character(),
##   visit_date = col_date(format = "")
## )
```

```
cleaned %>% anti_join(visits)
```

```
## Joining, by = "visit_id"
```

```
## # A tibble: 4 x 4
##   visit_id visitor quantity reading
##      <dbl> <chr>   <chr>      <dbl>
## 1      752 lake    rad         2.19
## 2      752 lake    sal         0.09
## 3      752 lake    temp      -16
## 4      752 roe     sal         0.416
```

Are there any sites in `visits` that don't have matches in `cleaned`?

7. Join `visits` with the cleaned-up table of readings:

```
cleaned <- visits %>%
  dplyr::inner_join(cleaned, by = c("visit_id" = "visit_id"))
cleaned
```

```
## # A tibble: 14 x 6
##    visit_id site_id visit_date visitor quantity reading
##       <dbl> <chr>   <date>     <chr>   <chr>      <dbl>
##  1      619 DR-1    1927-02-08 dyer    rad         9.82
##  2      619 DR-1    1927-02-08 dyer    sal         0.13
##  3      622 DR-1    1927-02-10 dyer    rad         7.8
##  4      622 DR-1    1927-02-10 dyer    sal         0.09
##  5      734 DR-3    1930-01-07 pb      rad         8.41
##  6      734 DR-3    1930-01-07 lake    sal         0.05
##  7      734 DR-3    1930-01-07 pb      temp      -21.5
##  8      735 DR-3    1930-01-12 pb      rad         7.22
##  9      751 DR-3    1930-02-26 pb      rad         4.35
## 10      751 DR-3    1930-02-26 pb      temp      -18.5
## 11      837 MSK-4   1932-01-14 lake    rad         1.46
## 12      837 MSK-4   1932-01-14 lake    sal         0.21
## 13      837 MSK-4   1932-01-14 roe     sal         0.225
## 14      844 DR-1    1932-03-22 roe     rad        11.2
```

Join `visited.csv` with `site.csv` to get (date, latitude, longitude) triples for site visits.

7. Find the dates of the highest radiation reading at each site:

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(max_rad = max(reading)) %>%
  dplyr::filter(reading == max_rad)
```

```
## # A tibble: 3 x 7
## # Groups:   site_id [3]
##   visit_id site_id visit_date visitor quantity reading max_rad
##      <dbl> <chr>   <date>     <chr>   <chr>      <dbl>   <dbl>
## 1      734 DR-3    1930-01-07 pb      rad         8.41    8.41
## 2      837 MSK-4   1932-01-14 lake    rad         1.46    1.46
## 3      844 DR-1    1932-03-22 roe     rad        11.2    11.2
```

Another way to do it:

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::top_n(1, reading) %>%
  dplyr::select(site_id, visit_date, reading)
```

```
## # A tibble: 3 x 3
## # Groups:   site_id [3]
##   site_id visit_date reading
##   <chr>   <date>       <dbl>
## 1 DR-3    1930-01-07    8.41
## 2 MSK-4   1932-01-14    1.46
## 3 DR-1    1932-03-22   11.2
```

Explain why this ***doesn't*** *work.*

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::summarize(max_rad = max(reading)) %>%
  dplyr::ungroup() %>%
  dplyr::filter(reading == max_rad)
```

```
## Error: object 'reading' not found
```

8. Normalize radiation against the highest radiation seen per site:

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(
    max_rad = max(reading),
    frac_rad = reading / max_rad) %>%
  dplyr::select(visit_id, site_id, visit_date, frac_rad)
```

```
## # A tibble: 7 x 4
## # Groups:   site_id [3]
##   visit_id site_id visit_date frac_rad
##      <dbl> <chr>   <date>        <dbl>
## 1      619 DR-1    1927-02-08    0.873
## 2      622 DR-1    1927-02-10    0.693
## 3      734 DR-3    1930-01-07    1
## 4      735 DR-3    1930-01-12    0.859
## 5      751 DR-3    1930-02-26    0.517
## 6      837 MSK-4   1932-01-14    1
## 7      844 DR-1    1932-03-22    1
```

Normalize salinity against mean salinity by site.

9. Find stepwise change in radiation per site by date:

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(delta_rad = reading - dplyr::lag(reading)) %>%
  dplyr::arrange(site_id, visit_date)
```

```
## # A tibble: 7 x 7
## # Groups:   site_id [3]
##   visit_id site_id visit_date visitor quantity reading delta_rad
##      <dbl> <chr>   <date>     <chr>   <chr>      <dbl>     <dbl>
## 1      619 DR-1    1927-02-08 dyer    rad         9.82     NA
## 2      622 DR-1    1927-02-10 dyer    rad         7.8      -2.02
```

```
## 3      844 DR-1    1932-03-22 roe     rad        11.2       3.45
## 4      734 DR-3    1930-01-07 pb      rad         8.41     NA
## 5      735 DR-3    1930-01-12 pb      rad         7.22     -1.19
## 6      751 DR-3    1930-02-26 pb      rad         4.35     -2.87
## 7      837 MSK-4   1932-01-14 lake    rad         1.46     NA
```

Find length of time between visits by site.

10. Find sites that experience any stepwise increase in radiation between visits:

```
cleaned %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(delta_rad = reading - dplyr::lag(reading)) %>%
  dplyr::filter(!is.na(delta_rad)) %>%
  dplyr::summarize(any_increase = any(delta_rad > 0)) %>%
  dplyr::filter(any_increase)
```

```
## # A tibble: 1 x 2
##   site_id any_increase
##   <chr>   <lgl>
## 1 DR-1    TRUE
```

Find sites with visits more than one year apart.

5.3 CREATING CHARTS

We will use data on the mass and home range area (HRA) of various species from:

> Tamburello N, Côté IM, Dulvy NK (2015) Data from: Energy and the scaling of animal space use. Dryad Digital Repository. https://doi.org/10.5061/dryad.q5j65

```
hra <- readr::read_csv(here::here("data", "home-range-database.csv"))
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   mean.mass.g = col_double(),
##   log10.mass = col_double(),
##   mean.hra.m2 = col_double(),
##   log10.hra = col_double(),
##   preymass = col_double(),
##   log10.preymass = col_double(),
##   PPMR = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

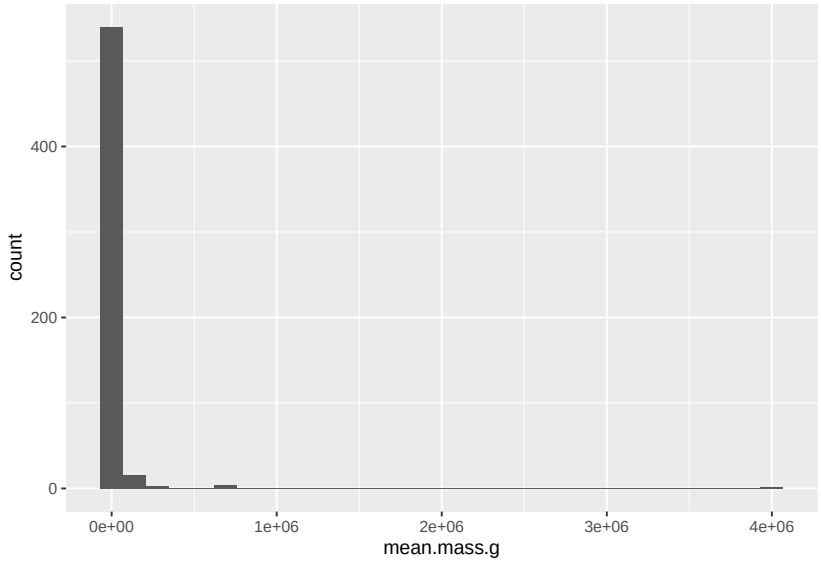
```
head(hra)
```

```
## # A tibble: 6 x 24
##   taxon common.name class order family genus species primarymethod N
##   <chr> <chr>       <chr> <chr> <chr>  <chr> <chr>   <chr>         <chr>
## 1 lake~ american e~ acti~ angu~ angui~ angu~ rostra~ telemetry     16
## 2 rive~ blacktail ~ acti~ cypr~ catos~ moxo~ poecil~ mark-recaptu~ <NA>
## 3 rive~ central st~ acti~ cypr~ cypri~ camp~ anomal~ mark-recaptu~ 20
## 4 rive~ rosyside d~ acti~ cypr~ cypri~ clin~ fundul~ mark-recaptu~ 26
## 5 rive~ longnose d~ acti~ cypr~ cypri~ rhin~ catara~ mark-recaptu~ 17
## 6 rive~ muskellunge acti~ esoc~ esoci~ esox  masqui~ telemetry     5
## # ... with 15 more variables: mean.mass.g <dbl>, log10.mass <dbl>,
## #   alternative.mass.reference <chr>, mean.hra.m2 <dbl>, log10.hra <dbl>,
## #   hra.reference <chr>, realm <chr>, thermoregulation <chr>,
## #   locomotion <chr>, trophic.guild <chr>, dimension <chr>,
## #   preymass <dbl>, log10.preymass <dbl>, PPMR <dbl>,
## #   prey.size.reference <chr>
```

1. Look at how mass is distributed:

```
ggplot2::ggplot(hra, mapping = aes(x = mean.mass.g)) +
  ggplot2::geom_histogram()
```

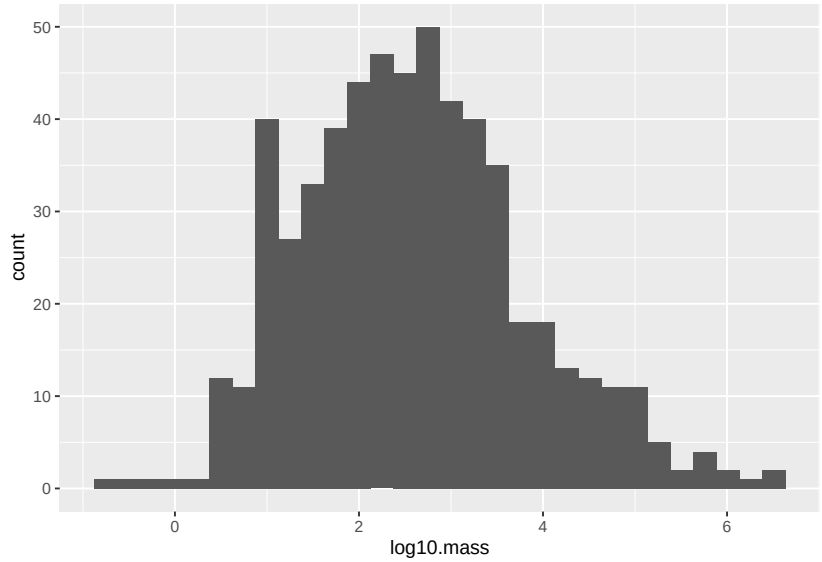
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Try again with `log10.mass`:

```
ggplot2::ggplot(hra, mapping = aes(x = log10.mass)) +
  ggplot2::geom_histogram()
```

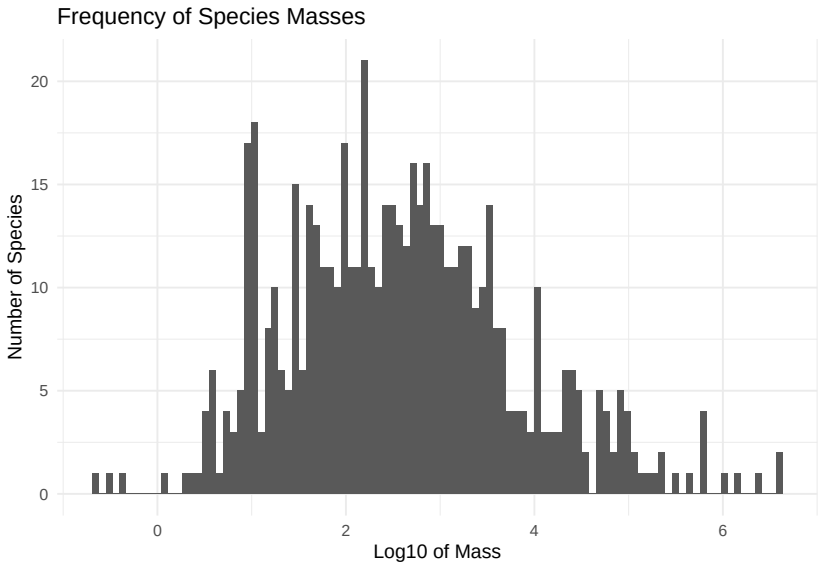
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Create histograms showing the distribution of home range area using linear and log scales.

2. Change the visual appearance of a chart:

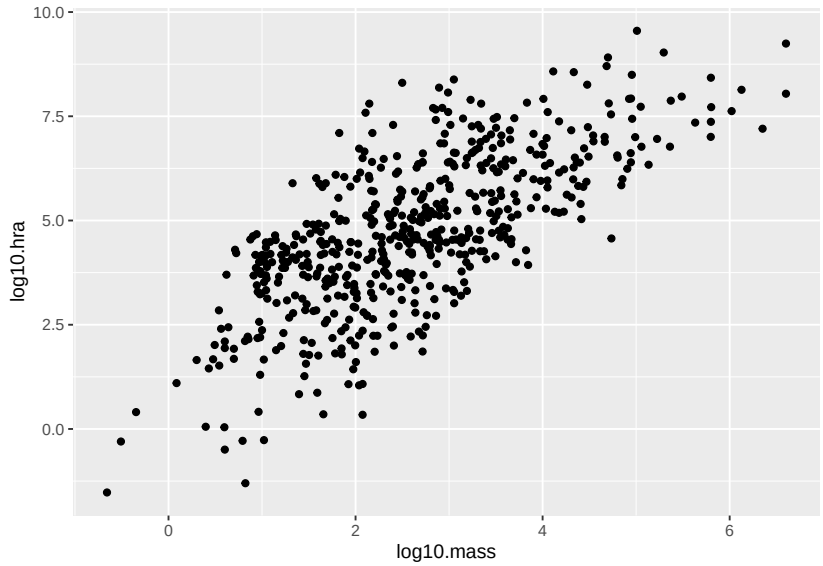
```
ggplot2::ggplot(hra, mapping = aes(x = log10.mass)) +
  ggplot2::geom_histogram(bins = 100) +
  ggplot2::ggtitle("Frequency of Species Masses") +
  ggplot2::xlab("Log10 of Mass") +
  ggplot2::ylab("Number of Species") +
  ggplot2::theme_minimal()
```



Show the distribution of home range areas with a dark background.

3. Create a scatterplot showing the relationship between mass and home range area:

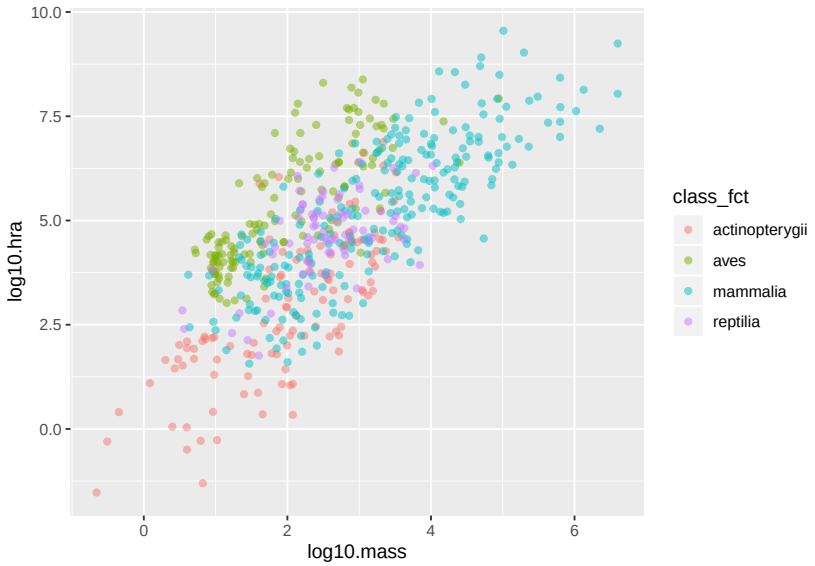
```
ggplot2::ggplot(hra, mapping = aes(x = log10.mass, y = log10.hra)) +
  ggplot2::geom_point()
```



Create a similar scatterplot showing the relationship between the raw values rather than the log values.

4. Colorize scatterplot points by class:

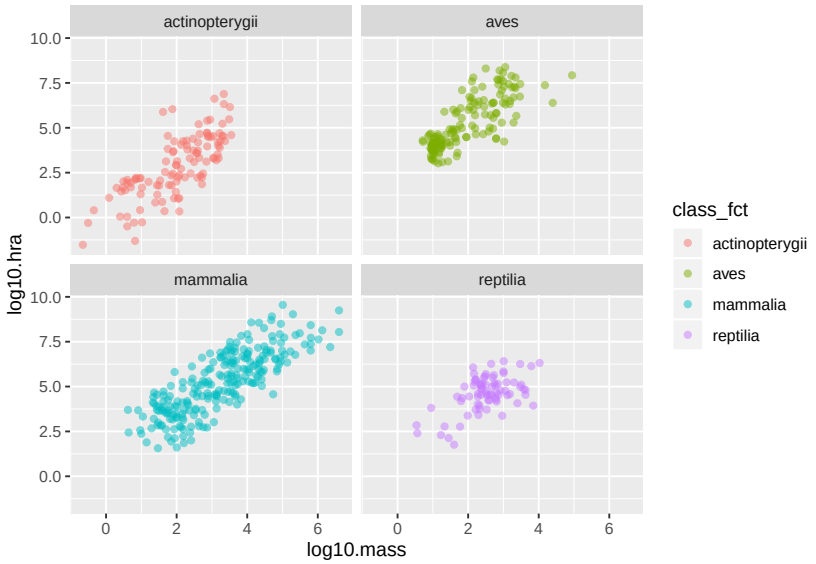
```
hra %>%
  dplyr::mutate(class_fct = as.factor(class)) %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra, color = class_fct)) +
  ggplot2::geom_point(alpha = 0.5)
```



Group by order and experiment with different alpha values.

5. Create a faceted plot:

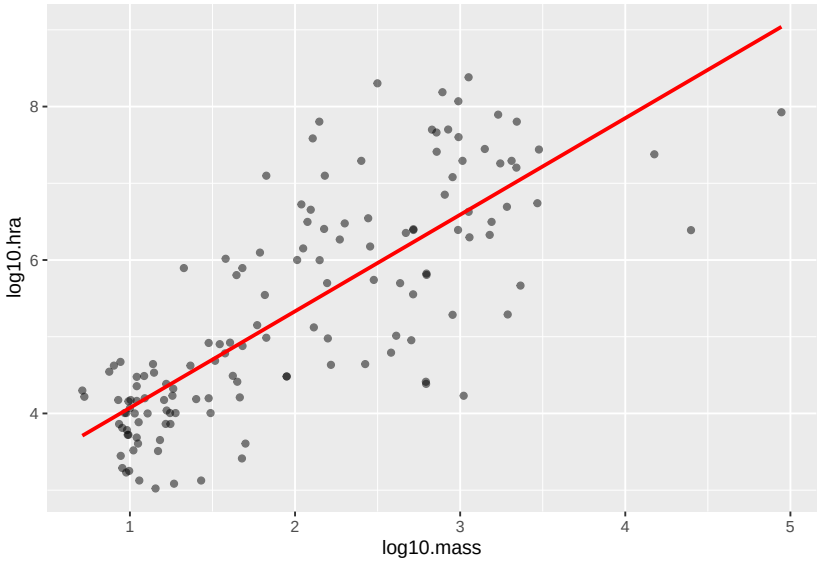
```
hra %>%
  dplyr::mutate(class_fct = as.factor(class)) %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra, color = class_fct)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::facet_wrap(vars(class_fct))
```



Create a plot faceted by order for just the reptiles.

6. Fit a linear regression to the logarithmic data for birds:

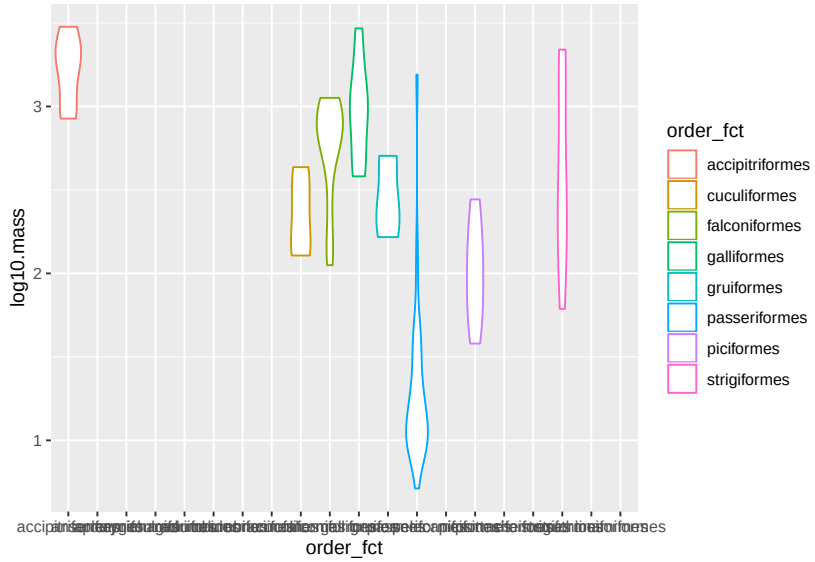
```
hra %>%
  dplyr::filter(class == "aves") %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::geom_smooth(method = lm, color = 'red', se = FALSE)
```



Fit a line to the raw data for birds rather than the logarithmic data.

7. Create a violin plot of mass by order for birds:

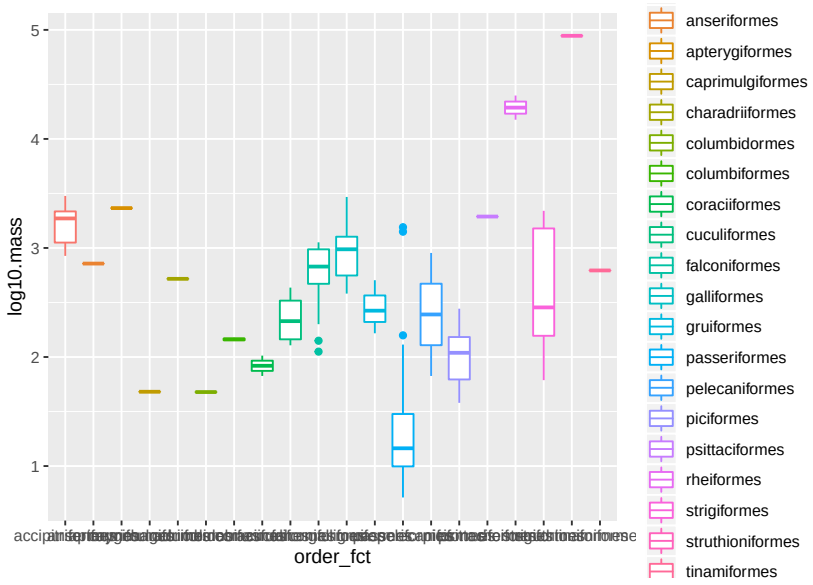
```
hra %>%
  dplyr::filter(class == "aves") %>%
  dplyr::mutate(order_fct = as.factor(order)) %>%
  ggplot2::ggplot(mapping = aes(x = order_fct, y = log10.mass, color = order_fct)) +
  ggplot2::geom_violin()
```



Rotate the labels on the X axis to make this readable, then explain the gaps.

8. Display the same data as a boxplot:

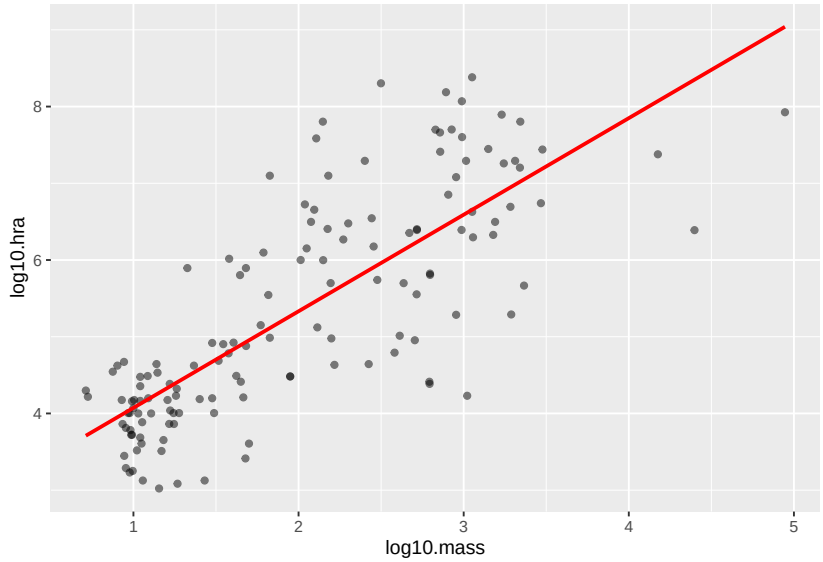
```
hra %>%
  dplyr::filter(class == "aves") %>%
  dplyr::mutate(order_fct = as.factor(order)) %>%
  ggplot2::ggplot(mapping = aes(x = order_fct, y = log10.mass, color = order_fct)) +
  ggplot2::geom_boxplot()
```



Fix the labels and remove orders that only contain one species.

9. Save the linear regression plot for birds as a PNG:

```
hra %>%
  dplyr::filter(class == "aves") %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::geom_smooth(method = lm, color = 'red', se = FALSE)
```



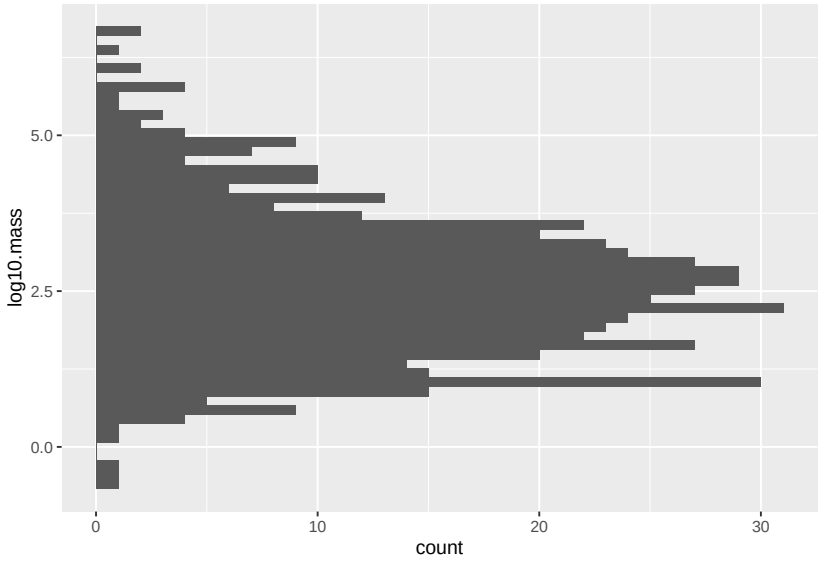

```
ggsave(here::here("birds.png"))
```

```
## Saving 6.5 x 4.5 in image
```

Save the plot as SVG scaled to be 8cm wide.

10. Create a horizontal histogram with 50 bins:

```
ggplot2::ggplot(hra, mapping = aes(x = log10.mass)) +
  ggplot2::geom_histogram(bins = 50) +
  ggplot2::coord_flip()
```



Use `stat_summary` *to summarize the relationship between mass and home range area by class.*

6 Reproducibility

There are several key cornerstones for doing rigorous and sound scientific research, two of which are reproducibility and replicability (Patil et al., 2016). Replicability is when a study is repeated by other independent research groups. Reproducibility is when, given the same data and the same analytical/computational steps, a scientific result can be verified. Both of these concepts are surprisingly difficult to achieve.

This course is about data analysis, so we'll be focusing solely on reproducibility rather than replicability. At present, there is little effort in science for having research be reproducible, likely due in many ways to a lack of training and awareness. Being reproducible isn't just about doing better science, it can also:

1. Make you much more efficient and productive, as you spend less time between coding and putting your results in the document.
2. Make you more confident in your results, since what you report and show as figures or tables will be exactly what you get from your analysis. No copying and pasting required!

There are many aspects to reproducibility, such as:

- Organized files and folder, preferably based on a standard or conventional structure.
- Understandable and readable code that is documented and descriptive.
- Results from analyses are identical to results presented in scientific output (e.g. article, poster, slides).
- Results from analyses are identical when code is executed on other machines (results aren't dependent on one computer).
- Explicit description or instruction on the order that code and scripts need to be executed.

We'll cover the first three items in this course.

6.1 PROJECT ORGANIZATION

First off, what exactly does "project" mean? That depends a bit on the group, individual, or situation, but for our purposes, a "project" is anything related to one or more completed scientific "products" (e.g. poster, slides, manuscript, package, teaching material) related to a specific question or goal. This could be "one manuscript publication and associated conference presentations" per project. Confining a project to one "scientific output" facilitates keeping the project reproducible is kept reproducible, all files will relate to that "output", and can be easily archived once the manuscript has been published. However, this definition could be different depending on your own situation and goals.

The ability to read, understand, modify, and write simple pieces of code is an essential skill for modern data analysis tasks and projects. Here we introduce you to some of the best practices one should have while writing their code. , many of which were taken from published "best practices" articles (Noble, 2009; Taschuk and Wilson, 2017; Wilson et al., 2017).

- Organise all R scripts and files in a single parent directory using a common and consistent folder and file structure.
- Use version control to track changes to files.
- Make raw data "read-only" (don't edit it directly) and use code to show what was done.
- Write and describe code for people to read by being descriptive and using a style guide.
- Think of code as part of your research product: Write for an audience or other reader.
- Create functions to avoid repetition.
- Whenever possible, use code to create output (figures, tables) rather than manualling creating or editing them.

Managing your projects in a reproducible fashion doesn't just make your science reproducible, it also makes your life easier! RStudio is here to help us with that by using R Projects. RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.

It is strongly recommended that you store *all* the necessary files that will be used in your code in the **same parent directory**. You can then use relative file paths to access them (we'll talk about file paths below). This makes the directory and R Project a "product" or "bundle/package". Like a tiny machine, that needs to have all its parts in the same place.

6.1.1 CREATING YOUR FIRST PROJECT

There are many ways one could organise a project folder. We'll set up a project directory folder using the prodigenr package:

```
# prodigenr::setup_project("ProjectName")
prodigenr::setup_project("learningr")
```

(You can also create a new project in RStudio by using "File -> New Project ->"Scientific Analysis Project using prodigenr".)

When we use the `::` colon here, we are telling R "use setup_project function from the prodigenr package". This function will then create the following folders and files:

```
learningr
  R
      README.md
      fetch_data.R
      setup.R
  data
      README.md
  doc
      README.md
  .gitignore
  DESCRIPTION
  learningr.Rproj
  README.md
```

This forces a specific, and consistent, folder structure to all your work. Think of this like the "introduction", "methods", "results", and "discussion" sections of your paper. Each project is then like a single manuscript or report, that contains everything relevant to that specific project. There is a lot of power in something as simple as a consistent structure. Projects are used to make life easier. Once a project is opened within RStudio the following actions are taken:

- A new R session (process) is started.
- The current working directory is set to the project directory.
- RStudio project options are loaded.

The README in each folder explains a bit about what should be placed there. But briefly:

1. Documents like manuscripts, abstracts, and exploration type documents should be put in the `doc/` directory (including R Markdown files which we will cover later).
2. Data, raw data, and metadata should be in either the `data/` directory or in `data-raw/` for the raw data.
3. All R files and code should be in the `R/` directory.
4. Name all new files to reflect their content or function. Follow the tidyverse style guide for file naming.

Note the `DESCRIPTION` file. This is used as metadata about the project and is useful when working on R projects. For any project, it is **highly recommended** to use version control. We'll be covering version control in more detail later in the course.

6.1.2 EXERCISE: BETTER FILE NAMING

Look at the list of file names below. Which file names are good names and which shouldn't you use?

```
fit models.R
fit-models.R
foo.r
stuff.r
get_data.R
Manuscript version 10.docx
manuscript.docx
new version of analysis.R
trying.something.here.R
plotting-regression.R
utility_functions.R
code.R
```

6.1.3 SHOULD YOU KEEP YOUR DATA UNDER VERSION CONTROL?

We have a `data/` folder for a reason. But you might not want to keep the data under version control, for several reasons:

1. It's a large dataset (tens or more Mb in file size)
2. There are sensitive and/or personally-identifying information in the data

As a rule of thumb, if you can send the data by an email attachment, you could probably put it into Git. Unless there is sensitive or personal data, then don't. If it isn't kept under version control, make sure you include a reference to how or where you got the data, either as an R script showing the code you used to import/clean/download it or described in the `README.md` file.

6.2 REUSABILITY

Part of reproducibility is also making sure your scripts and file organization is "reusable" meaning that others (or yourself) can run it again. So, for instance, making sure to use "relative file paths" compared to "absolute file paths" (we'll cover these in a bit). Or indicating which other R packages your code depends on. So here we'll cover how to make sure your scripts and project files are reusable.

6.2.1 KEEPING A CLEAN SLATE

When you finish writing your R code for the day and close the session, you probably will be asked about saving your session. What this does is everything kept in the environment (e.g. all objects, functions, or datasets you created and used during the session) get saved to an `.RData` file. Then, the next time you open up your R session, R will see this `.RData` file and load everything in that file. Everything you did previously will be loaded into your environment. This seems like a good thing… but it's not. Imagine eating your dinner on a really dirty plate… that's not pleasant right? Loading a previous session is like that dirty plate.

So, to make sure you always use a clean slate, we'll run a handy function from the usethis package to stop R from saving and loading this `.RData` file, ensuring you have a clean working environment. You only need to run this function once, as it will set the appropriate RStudio settings for you.

```
usethis::use_blank_slate()
```

We'll use the usethis package more throughout this chapter and others, as it provides several very useful functions when working with projects.

6.2.2 PACKAGES, DATA, AND FILE PATHS

A major strength of R is in its ability for others to easily create packages that simplify doing complex tasks (e.g. creating figures with the ggplot2 package) and for anyone to easily install and use that package[1]. You load a package by writing:

```
library(tidyverse)
```

Working with multiple R scripts and files, it quickly gets tedious to always write out each library function at the top of each script. A better way of managing this is to create a new file, keep all package loading code in that file, and sourcing that file in each R script. So, to create a new R file in the `R/` folder, we'll use this `use_r()` function from the usethis package:

```
usethis::use_r("package-loading")
```

This creates a file called `package-loading.R` in the `R/` folder. In the `package-loading.R` file, add this code to it.

```
library(tidyverse)
```

Then in other R scripts in the `R/`, include this code at the top the script:

```
source(here::here("R/package-loading.R"))
```

The here package uses a function called `here()` that makes it easier to manage file paths. What is a file path and why is this necessary? A file path is the list of folders a file is found in. For instance, your resume may

[1] You may encounter some who say you shouldn't rely on packages and to only use base R functions. However, this is seriously bad advice since the ecosystem of R packages can greatly simplify your life doing data analysis. Plus, packages greatly expand and enhance the capability of R, so make use of packages! If someone invents a wheel, why wouldn't you use it?

be found in `/Users/Documents/personal_things/resume.docx`. The problem with file paths in R is that when you run a script interactively (e.g. what we do in class and normally), the file path is located at the Project level (where the `.Rproj` file is found). You can see the file path by looking at the top of the "Console".

But! When you `source()` an R script, it may likely run *in the folder it is saved in*, e.g. in the `R/` folder. So your file path `R/packages-loading.R` won't work because there isn't a folder called R in the `R/` folder. Often people use the function `setwd()`, but this is *never* a good idea since using it makes your script *runnable only on your computer*... which makes it no longer reproducible. We use the `here()` function to tell R to go to the project root (where the `.Rproj` file is found) and then use the file path from that point. This simple function can make your work more reproducible and easier for you to use later on.

We also use the `here()` function when we import a dataset or save a dataset. So, let's load in the NYC Dog License dataset. First, save the CSV in the `data/` folder. Then create a new R file:

```
usethis::use_r("load-data")
```

And write these lines in the file:

```
source(here::here("R/package-loading.R"))
dog_license <- read_csv(here::here("data/nyc-dog-licenses.csv.gz"))
head(dog_license)
```

That is how we will load data in from now on.

Here are a few other tips for keeping your code reusable:

- When encountering a difficult problem, try to find R packages or functions that do your problem for you[2].
- Split up your analyses steps into individual files (e.g. "model" file, "plot" file). Then `source` those files as needed or save the output in `data/` to use it in other files.

[2]You may hear some people say "oh, don't bother with R packages, do everything in base R"... don't listen to them. Do you build a computer from scratch everytime you want to do any type of work? Or a car when you want to drive somewhere? No, you don't. Make use of other people's hard work to make tools that simplify your life.

- Try not to have R scripts be too long (e.g. more than 500 lines of code). Keep script sizes as small as necessary and as specific as possible (have a single purpose). A script should have an end goal.

6.3 READABILITY

There are two reasons we write code: to instruct the computer to do something and to record the steps we took to get a particular result for us or others to understand. For computers, *how* or *what* you write doesn't matter, as long as the code is correct. Computers don't need to *understand* the code. But humans do need to understand it. We need clear language and explicit meaning in order to understand what is going on. Humans write code, humans read code, and humans must maintain it and fix any errors. So, what you write and how you write it is extremely important.

Like natural human languages, R has a relaxed approached to how R code is written. This has some nice advantages, but also some major disadvantages, notably that writing styles can be quite different across the world or even within one's own code. So, it's important to stick to some guidelines, for instance, as laid out by the tidyverse style guide. Some other tips include:

- Write your code assuming other people will be reading it.
- Stick to a *style guide*. (We're repeating this because it's really important!)
- Use full and descriptive words when typing and creating objects.
- Use white space to separate concepts (empty lines between them, use spaces, and/or tabs).
- Use RStudio R Script Sections (`"Code->Insert Section"` or `Ctrl-Shift-R`) to separate content in scripts.

Even though R doesn't care about naming, spacing, and indenting, it really matters how your code looks. Coding is just like writing. Even though you may go through a brainstorming note-taking stage of writing, you eventually need to write correctly so others can understand, *and read*, what you are trying to say. Brainstorming and exploratory work is fine, but eventually you need to write code that will be legible. That's why using a style guide is really important.

6.3.1 EXERCISE: MAKE THE CODE MORE READABLE

Using the style guide found in the link, try to make these code more readable. Edit the code so it follows the correct style and so it is easier to understand and read. You don't need to understand what the code does, just follow the guide.

```
# Object names
DayOne
dayone
T <- FALSE
c <- 9
mean <- function(x) sum(x)

# Spacing
x[,1]
x[ ,1]
x[ , 1]
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
function (x) {}
function(x){}
height<-feet*12+inches
mean(x, na.rm=10)
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10

# Indenting
if (y < 0 && debug)
message("Y is negative")
```

FIXME: The below "details" will need to be dealt with since PDF doesn't allow this FIXME: Maybe move to a solutions section at the end of chapter?

Click for a possible solution

The old code is in comments and the better code is below it.

```
# Object names

# Should be camel case
```

```r
# DayOne
day_one
# dayone
day_one

# Should not over write existing function names
# T = TRUE, so don't name anything T
# T <- FALSE
false <- FALSE
# c is a function name already. Plus c is not descriptive
# c <- 9
number_value <- 9
# mean is a function, plus does not describe the function which is sum
# mean <- function(x) sum(x)
sum_vector <- function(x) sum(x)

# Spacing
# Commas should be in correct place
# x[,1]
# x[ ,1]
# x[ , 1]
x[, 1]
# Spaces should be in correct place
# mean (x, na.rm = TRUE)
# mean( x, na.rm = TRUE )
mean(x, na.rm = TRUE)
# function (x) {}
# function(x){}
function(x) {}
# height<-feet*12+inches
height <- feet * 12 + inches
# mean(x, na.rm=10)
mean(x, na.rm = 10)
# sqrt(x ^ 2 + y ^ 2)
sqrt(x^2 + y^2)
# df $ z
df$z
# x <- 1 : 10
x <- 1:10

# Indenting should be done after if, for, else functions
# if (y < 0 && debug)
```

```
# message("Y is negative")
if (y < 0 && debug)
    message("Y is negative")
```

6.3.2 AUTOMATIC STYLING WITH STYLER

You may have organised the exercise by hand, but it's possible to do it automatically. The tidyverse style guide has been implemented into the styler package to automate the process of following the guide by directly re-styling selected code. The styler snippets can be found in the Addins function in the RStudio "Addins" menu after you have installed it.

RStudio also has its own automatic styling ability, through the menu item `"Code -> Reformat Code"` (or `Ctrl-Shift-A`). Try both methods of styling on the exercise code above. There are slight differences in how each method works and they both aren't always perfect. For now, let's stick with using the styler package. The styler functions work on R code within both `.R` script files as well as R code within `.Rmd` documents, which we will cover later in this lesson.

There are several styler RStudio addins, but we'll focus on the two:

- `"Style selection"`: Highlight text and click this button to reformat the code.
- `"Style active file"`: Code in the `.R` or `.Rmd` file you have open and visible in RStudio will be reformatted when you click this button.

There are two other styler functions that are also useful:

- `styler::style_file("path/to/filename")`: Styles the whole file as indicated by the file path in the first argument. Can be either an `.R` or `.Rmd` file.
- `styler::style_dir("directoryname")`: Styles all files in the indicated directory in the first argument.

Let's try the `styler::style_file()` function out. Inside a file called `non-styled-code.R`, it has:

```
# Spacing
x[,1]
mean (x, na.rm = TRUE)
function (x) {}
height<-feet*12+inches
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10

# Indenting
if (y < 0 && debug)
message("Y is negative")
```

Then we run:

```
styler::style_file("testing-styler.R")
```

```
Styling  1  files:
 testing-styler.R

Status  Count   Legend
    0   File unchanged.
    1   File changed.
    0   Styling threw an error.

Please review the changes carefully!
```

Which changes the file to be styled!

```
# Spacing
x[, 1]
mean(x, na.rm = TRUE)
function(x) {}
height <- feet * 12 + inches
sqrt(x^2 + y^2)
df$ z
x <- 1:10

# Indenting
if (y < 0 && debug) {
```

```
  message("Y is negative")
}
```

This is more or less everything that the styler package does.

6.3.3 EXERCISE: USE STYLER TO FIX CODE FORMATTING

Use the styler package function on the code from the previous exercise by either running `styler::style_file()` or with the `"Style selection"` addin when highlighting the code.

6.4 INTEGRATING TEXT, CODE, AND RESULTS

The most obvious demonstration of reproducibility is when the results obtained from executing the analysis code (by an independent entity) are indentical to the results presented in the scientific output such as in an article. When there is agreement between these two, reproducibility has been more or less acheived. In R, there are tools available to *completely* ensure that this happens by directly inserting the results from the code *into the scientific ouput*. This is done by using R Markdown, which interweaves R code with text. So instead of, for example, manually inserting a figure, you write R code within the document to insert the figure for you! Using R Markdown can save so much time and get your work that much closer to being reproducible.

There are many other advantages to using R Markdown. From the single R Markdown format you can use it to create manuscripts, posters, slides, websites, books, and many more from simply using R Markdown. In fact, this book was written using R Markdown. As a bonus, switching between citation formats or Word templates for different journals is easier than doing it with Word.

6.4.1 MARKDOWN

R Markdown uses, well, Markdown as the format to convert to multiple document types. Fun fact: This website is built based on Markdown! While there are many "flavours" of Markdown that have been developed over the years, R Markdown uses the Pandoc version. Pandoc is a combination of pan which is Latin for "all" and doc which means document.

Markdown is a "markup language" meaning that special characters mean certain things, which we will cover below.

To format text, such as to bold or make a list, you use the special characters. You write Markdown as plain text (like R code), so you don't need any special software (like you do with Word documents). Most features needed for writing a scientific document are available in Markdown, but not all. *Tip*: Try to fit your writing and document creation around what Markdown can acheive, rather than force or fight Markdown to do something it wasn't designed to do.

All right, let's create and save an R Markdown file. In RStudio, go to `File -> New File -> R Markdown`. A dialog box will pop up. In the "Title" section, type in `Reproducible documents` and in the "Author" section type in your name. Choose the HTML output format. Save this file as `learning-rmarkdown.Rmd` in the `doc/` folder.

Inside the file, there is a bunch of text that shows some basic formatting you can use for writing Markdown. For now, delete everything *except* the top part of the file (the part surrounded by `---`). This part is called the YAML header, which we will cover more a bit later. Try converting the file to HTML by hitting the "Knit" button at the top or by typing out `Ctrl-Shift-K`.

FIGURE 6.1 "Knit" button.

6.4.1.1 Headers

Creating headers (like chapters or sections) is indicated by using one or more `#` at the beginning of a line, prefixing some text:

```
# Header level 1

Paragraph.
```

```
## Header level 2

Paragraph.

### Header level 3

Paragraph.
```

This creates the section headers as seen directly above ("Headers") or below ("Text formatting"). The header text *must* be on one line, otherwise the next line is interpreted as paragraph text.

See how it looks by "Knitting" the document ("Knit" button or `Ctrl-Shift-K`).

6.4.1.2 Text formatting

To format text individually, surround the text with the special characters, as shown here:

- `**bold**` gives **bold**.
- `*italics*` gives *italics*.
- `super^script^` gives super$^{\text{script}}$.
- `sub~script~` gives sub$_{\text{script}}$.

What if you want to use the special character as simple text? Prefix it with an \, so `\*` becomes *, `\^` becomes ^, and `\~` becomes ~.

6.4.1.3 Lists

To create an unnumbered list, do:

```
- item 1
- item 2
- item 3
```

which gives...

- item 1

- item 2
- item 3

Notice the empty lines above and below the line, those are important. To create a numbered list, do:

```
1. item 1
2. item 2
3. item 3
```

which gives...

1. item 1
2. item 2
3. item 3

See how it looks by "Knitting" the document ("Knit" button or `Ctrl-Shift-K`).

6.4.1.4 Blockquotes

Sometimes (probably not too commonly), you may need to quote someone by using "blockquotes". To do that, do:

```
> Blockquote
```

which gives...

> Blockquote

Blockquotes can be as many lines as you want. To stop a paragraph from being in the blockquote, separate the text with an empty line:

```
> Bockquote paragraph

Regular paragraph
```

6.4.1.5 Adding footnotes

Footnotes can be added by using `[^some-text-label]`, such as:

```
Footnote[^1]

[^1]: Footnote content.
```

which gives...

Footnote[3]

So you can write some text, add some footnotes within, and include the footnote content right below the paragraph:

```
Paragraph text[^1], with some more text[^reference].

[^1]: This is the first footnote.
[^reference]: This is the next footnote.

More paragraphs.
```

Notice the empty lines in between. The footnote should also be on one line, though it isn't strictly necessary. See how it looks by "Knitting" the document ("Knit" button or `Ctrl-Shift-K`).

6.4.1.6 Inserting pictures, images, or figures

You can include externally created (i.e. not by an R code chunk, discussed later on) png, jpeg, or pdf image file by adding (:

```
![Image caption here.](path/to/image/file.png)
```

So something like this:

```
![Steps to being more reproducible. Source DOI: 10.1038/d41586-018-05990-5](figures/r-repr
```

which gives...

[3] Footnote content.

FIGURE 6.2 Steps to being more reproducible. Source DOI: 10.1038/d41586-018-05990-5

Tip: Can also include links to images from the Internet, as a URL link.

If you want to modify the width or sizing, append something like `{width=##%}` to the end of the image insertion:

```
![Caption.](figures/r-reproducibility/code-sharing-steps.png){width=50%}
```

which gives...

FIGURE 6.3 Caption.

6.4.1.7 Adding links to websites

And a link can be linked in the following format:

```
[Link to GitHub](https://github.com).
```

gives...

Link to GitHub.

The above form is great for one time use, but what if you want to use the same link again later on? Use this form then:

```
Use multiple [links] in your document. The same [links] can be used again.

[links]: https://github.com
```

which gives...

Use multiple links in your document. The same links can be used again.

6.4.1.8 Inserting (simple) tables

You can insert tables with Markdown too. We wouldn't recommend doing it for complicated tables though, as it can get tedious fast! (A recommended approach for more complex or bigger tables is to make the table contents as a data frame in R first and then use the `knitr::kable()` function to create the table, as we'll cover in the R Markdown section below). You can even include Markdown text formatting inside the table:

```
|   | Fun | Serious |
|:--|--:|:--:|
| **Happy** | 1234 | 5678 |
| **Sad** | 123 | 456 |
```

which gives...

	Fun	Serious
Happy	1234	5678
Sad	123	456

The `|--:|` or `|:--|` tell the table to right-align or left-align, respectively, the values in the table column. Center-align is `|:--:|`. See how it looks by "Knitting" the document ("Knit" button or `Ctrl-Shift-K`).

6.4.1.9 Exercise: Try to re-create a document using Markdown

1. Open this link. This is a HTML file that has been created by using Markdown formatting.
2. Create a new R Markdown file, with output type "HTML", and save it as `mimic-html-file.Rmd`.

3. Delete all the automatically added text except the top part Inside the R Markdown file
4. Write text using Markdown formatting so that you can create a `html_document` that looks exactly like the linked file.
5. Knit the R Markdown document. Confirm that your version looks the same as the above version.

6.4.2 R MARKDOWN

R Markdown is an extension of Markdown that weaves together R code with Markdown formatted text all together in a single document. Output from the R code gets inserted directly into the document for a seamless integration of document writing and analysis reporting.

6.4.2.1 YAML header/metadata

Most Markdown documents (especially for R Markdown) include YAML metadata at the top of the document, surrounded by `---`. YAML is a data format, like CSV, that contains the metadata and various options that R Markdown uses for the entire document. Data in YAML is stored in the form `variable: value`. For instance, `title` or `author` is paired with their respective "values" and then used by Markdown when creating the document. Other options are also included here, such as what the converted output document should be, such as Word. There are many more output formats to choose from (e.g. slides, websites, books). The YAML header looks something like this:

```
---
title: "Document title"
author: Your Name
output: html_document
---
```

Here, there are three variable-value pairings: `title`, `author`, and `output`. In the `output` variable, the R Markdown function `html_document` is given so that the output document format is converted to HTML. There are also `word_document` and `pdf_document` settings. For now, we'll focus on the `html_document` output. Usually when you create the R Markdown file, this YAML header gets added automatically. There are additional options you can set in the output field, which we will cover later on.

6.4.2.2 Using R code chunks

R Markdown's primary function is to allow combining text and R code in the same document, which is incredibly powerful and useful! All R code chunks have the appearance:

````
```{r chunk-label-name, chunk.option="...", chunk.option=...}
...R code...
```
````

Notice that chunk options need to be on the same line (one single line). Any R code in the code chunk gets evaluated and any output gets inserted into the document. So if the code prints to the console, it will get inserted into the document.

The `r` tells R Markdown to run R on the code chunk, while the chunk label differentiates the chunk from other chunks. Using a chunk label also helps navigate a document when you use RStudio's "Document outline" (`Code -> Show Document Outline`). It also does a few other things.

Note: Standard practice is that code chunk labels should be named without `_`, spaces, or `.` and instead should be one word or be separated by `-`.

Let's load in some packages and data:

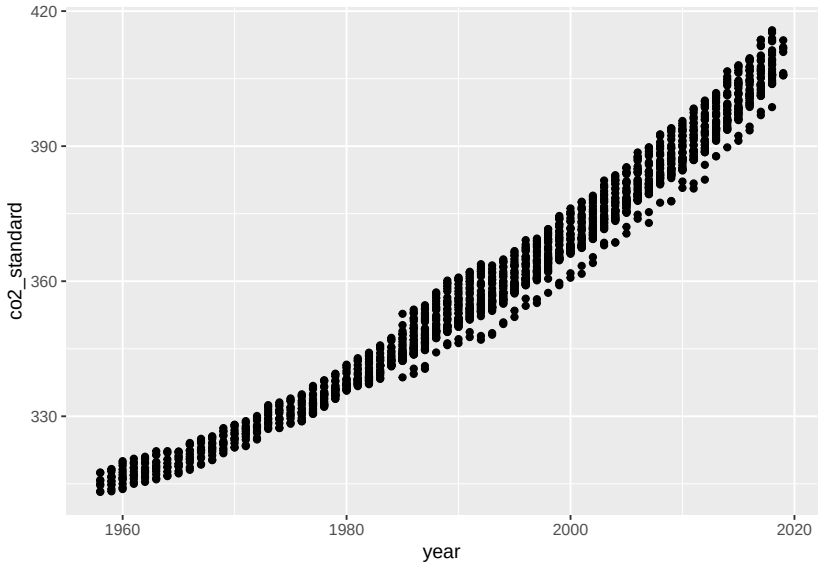
````
```{r setup}
library(tidyverse)
co2_data <- read_csv(here::here("data/co2.csv")) %>%
  filter(co2_standard > 0)
```
````

When running your code chunks interactively as you develop and write your document, R Markdown will look for a code chunk labeled `setup` to run first (for instance, to load all packages used in a document). Hence we name this chunk "setup".

6.4.2.3 Inserting figures

One of the most obvious benefits to using R Markdown is to automatically insert a plot. To do that we do:

```
```{r co2-time-plot}
ggplot(co2_data, aes(x = year, y = co2_standard)) +
    geom_point()
```
```



What if we want to include the plot but not the code? Easy! Set the chunk option `echo` to `FALSE`:

```
```{r co2-time-plot, echo=FALSE}
ggplot(co2_data, aes(x = year, y = co2_standard)) +
    geom_point()
```
```

Or what if you don't want R to run the code chunk, but still show the code? Set the chunk option `eval` (for evaluate) to `FALSE`:

```
```{r co2-time-plot, eval=FALSE}
ggplot(co2_data, aes(x = year, y = co2_standard)) +
    geom_point()
```
```

Since we have a figure, we can change some width, height, and alignment

options, as well as add a caption with the `fig.width`, `fig.height`, `fig.align`, and `fig.cap` (respectively):

````
```{r co2-time-plot, fig.cap="Add your figure title here.", fig.height=4, fig.width=7}
ggplot(co2_data, aes(x = year, y = co2_standard)) +
    geom_point()
```
````

```
ggplot(co2_data, aes(x = year, y = co2_standard)) +
    geom_point()
```

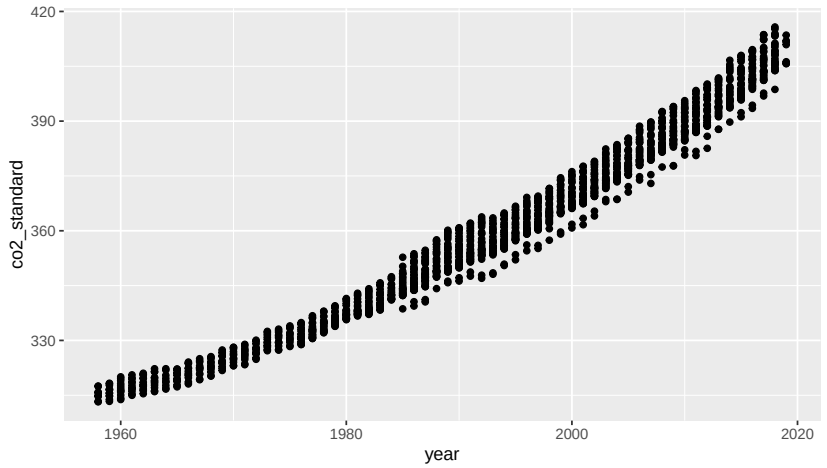


FIGURE 6.4 Add your figure title here.

What if we want to first run some code before running the plot, but don't want to include the output of the other code? Make sure that the output doesn't "print". So letting R evaluate both dataframe and plot, the code chunk will output and insert both into the knitted document.

````
```{r co2-time-plot-after-2000-1, fig.cap="Add your figure title here.", fig.height=4,
co2_after_2000 <- co2_data %>%
    filter(year >= 2000)

# Print dataframe
co2_after_2000
```
````

```
# Print plot
ggplot(co2_after_2000, aes(x = year, y = co2_standard)) +
    geom_point()
```

```
## # A tibble: 680 x 5
##     year month date_numeric co2_standard station
##    <dbl> <dbl>        <dbl>        <dbl> <chr>
##  1  2000     1        2000.         369. Mauna Loa, Hawaii, USA
##  2  2000     2        2000.         369. Mauna Loa, Hawaii, USA
##  3  2000     3        2000.         371. Mauna Loa, Hawaii, USA
##  4  2000     4        2000.         372. Mauna Loa, Hawaii, USA
##  5  2000     5        2000.         372. Mauna Loa, Hawaii, USA
##  6  2000     6        2000.         372. Mauna Loa, Hawaii, USA
##  7  2000     7        2001.         370. Mauna Loa, Hawaii, USA
##  8  2000     8        2001.         368. Mauna Loa, Hawaii, USA
##  9  2000     9        2001.         367. Mauna Loa, Hawaii, USA
## 10  2000    10        2001.         367. Mauna Loa, Hawaii, USA
## # ... with 670 more rows
```

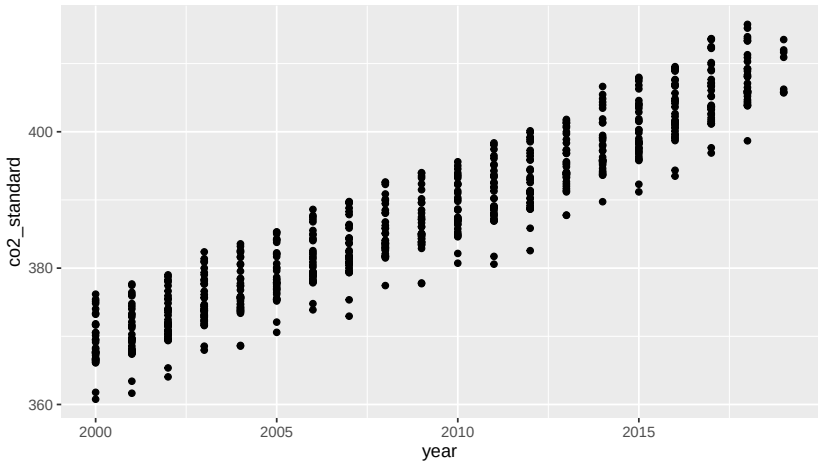


FIGURE 6.5 Add your figure title here.

Compare to this next code chunk, which will only output ("print") the plot. See how we don't include code that would send anything to the console to be "printed"? Only things that get "printed" will be included

in the R Markdown output document. Also notice how we renamed the chunk label? In R Markdown you can't have duplicate code chunk labels.

```
```{r co2-time-plot-after-2000-2, fig.cap="Add your figure title here.", fig.height=4,
co2_after_2000 <- co2_data %>%
    filter(year >= 2000)

# Print plot
ggplot(co2_after_2000, aes(x = year, y = co2_standard)) +
    geom_point()
```
```

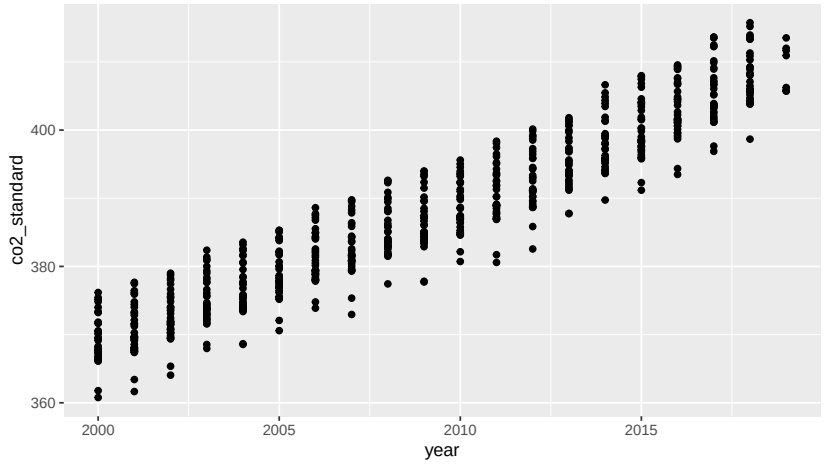


FIGURE 6.6 Add your figure title here.

6.4.2.4 Exercise: Add some figures to a R Markdown document

1. Create a new R Markdown file ("File -> New File -> R Markdown"), providing the title, author name (your name), and setting the output to HTML.
2. Save the file as `using-rmarkdown.Rmd` in the `doc/` folder.
3. Delete all text *except* the YAML header.
4. Create an R code chunk and call the label "setup". Write code so the packages and data are loaded.

TABLE 6.2
Table caption here.

month	Alert Station, NWT, Canada	Cape Grim, Tasmania, Australia	Mauna Loa, Hawaii, USA
1	378.54	364.63	354.61
2	380.06	364.65	355.96
3	380.75	364.73	356.11
4	381.07	364.89	357.47
5	380.36	363.34	356.57
6	378.43	363.70	356.64
7	371.33	364.16	354.50
8	365.98	364.60	352.51
9	365.11	364.85	350.85
10	369.21	364.90	351.49
11	374.77	364.80	352.24
12	376.94	364.68	353.53

5. Create another code chunk and call it "plot-licenses-by-year". Write R code to create a point plot (`geom_point()`) of the year on the `x` axis and number of licenses on the `y` axis.
6. Knit the document and see what the output looks like.
7. Change the theme of the plot to another builtin theme (*hint*: themes start with `theme_`).

6.4.2.5 Using R code chunks to insert tables

You can also create tables by using the `kable()` function from the knitr package. Let's create a table of the mean CO_2 concentration over the years at each monthly period for each station.

````
```{r mean-co2-table}
co2_data %>%
    select(station, month, co2_standard) %>%
    group_by(station, month) %>%
    summarise(MeanCO2 = round(mean(co2_standard, na.rm = TRUE), 2)) %>%
    spread(station, MeanCO2) %>%
    knitr::kable(caption = "Table caption here.")
```
````

6.4.2.6 Inline R code

Often you might have results inside the text you are writing. Here you can include R code within the text so that the results are inserted directly into the document. It looks like:

> The mean of CO_2 is ‘r round(mean(co2_data$co2_standard, na.rm = TRUE), 2)‘.

Which gives...

> The mean of CO_2 is `round(mean(co2_data$co2_standard, na.rm = TRUE), 2)`.

Keep in mind that inline R code can *only* insert a single number or character value, nothing more.

6.4.2.7 Citing literature with R Markdown

No scientific writing is complete without being able to include references. If you want to insert a citation, use the Markdown key `[@Cone2016]`, which will look like (Conery, 2016). The text `Cone2016` is the key that the bibliography manager uses to identify a specific reference. Adding more references is done by separating by a `;`, so like `[@AuthorYear; @Author2Year; @Author3Year]`. The resulting citation reference will be inserted at the bottom of the document. To get the bibliography to work, you'll also need to add a line to the YAML header like this:

```
---
title: "My report"
author: "Me!"
bibliography: my_references.bib
---
```

The `my_references.bib` is a `.bib` file found in the same folder as the `.Rmd` file. So in our case, the `.bib` file is in the `doc/` folder. You can also use other bibliography manager files, such as EndNote. See this documentation for which bibliography managers can be used.

Since all references are appended to the bottom of the document, it's good to add a final “Reference” section header to the end of your file, like so:

```
# References
```

6.4.2.8 Making your report prettier

This part mostly applies to HTML-based and PDF[4] outputs, since programmatically modifying or setting templates in Word documents is rather difficult[5]. Changing broad features of a document can be done by setting the "theme" of the document. Add an option in the YAML metadata like:

```
---
title: "My report"
output:
    html_document:
        theme: sandstone
---
```

Check out the R Markdown documentation for more types of themes you can use for HTML documents, and advanced topics such as parameterized R Markdown documents. Most of the Bootswatch themes are available for use in R Markdown to HTML conversion.

Want to add a Table of Contents? Easy! Add `toc: true` to the YAML header:

```
---
title: "My report"
output:
    html_document:
        theme: sandstone
        toc: true
---
```

Adding a `toc` only works for PDF and HTML, but *not* Word documents.

[4]Knitting to PDF requires LaTeX, which you can install from tinytex. After you install LaTeX you can create truly beautifully typeset PDF documents.

[5]If you really want to do it, the best way is to create your template in the `.odt`, and then convert to `.docx`.

6.4.2.9 Exercise: Add a summary table, inline results, and a prettier theme

1. Use the R Markdown file from the previous exercise (`using-rmarkdown.Rmd`).
2. Create three new header 1 `#` sections: Objective, Results, Conclusion.
3. Write in the "Objective" section an idea you have about the Dog License dataset. It can be as simple as "How many dogs are there in New Year City?". Include an *italics* in this sentence.
4. Create three new code chunks in the "Results" section: One for `setup`, one for `plot-dogs`, and one for `table-dogs`.
5. Write R code to load the packages and data in the `setup` chunk. Knit the document to see what it looks like.
6. Write R code to create a simple ggplot2 plot in the `plot-dogs` chunk related to your "Objective". Knit the document to see what it looks like.
7. Write R code to create a simple summary table using `kable()` in the `table-dogs` chunk related to your "Objective". Knit the document to see what it looks like.
8. Write an observation you made about the data from the plot and table in the "Conclusions" section. Include a **bold** text in this section.
9. Check out the Bootswatch themes and change your HTML theme to something else and add a Table of Contents.

6.5 KEY POINTS

- A structured and standard project folder and file layout is the first step to having a reproducible data analysis project.
- Writing documents in R Markdown can reduce the time spent on manual tasks since results can be easily re-generated and inserted into the final document to improve reproducibility.
- `usethis` has several helper functions for managing data analysis projects.
- Following a style guide and emphasizing readable code can lead to better quality code and to code that is more likely to be reproducible and reusable.
- Using Markdown to write documents is a great way to improve accessibility (since it is plain text only) and allows you to generate multiple types of output (HTML, PDF, slides, etc) from a single document source.

6.6 ADDITIONAL LEARNING RESOURCES AND MATERIAL

For learning:

- Use other programming languages in an R Markdown document.
- Online book for R Markdown
- R Markdown chapter in the R for Data Science book.

For help:

- RStudio helpful cheatsheets
- R Markdown cheatsheet (downloads a pdf file)
- R Markdown reference cheatsheet

Note: Source material for this chapter was modified from https://rqawr-course.lwjohnst.com/, as well as many other resources (see https://rqawr-course.lwjohnst.com/license/).

7 Data Manipulation

7.1 QUESTIONS

- How can I read tabular data into a program?
- How can I select subsets of my data?
- How can I calculate new values?
- How can I tell what's gone wrong in my programs?
- How can I operate on subsets of my data?
- How can I save my results?

7.2 MOTIVATION

FIXME: where is data introduced?

FIXME: clean version with no missing values and snake case column names, including month and year columns for some (all?) of the date columns.

FIXME: I'm assuming that at some point this data will live in R package, so we can delay importing data to end of chapter.

The RStudio Viewer has an interface much like other spreadsheet programs you might have used. You can use this Viewer to look at the `dog_licenses` tibble with the `View()` function:

```
View(dog_licenses)
```

This viewer has some basic data manipulation features:

- **Arrange** You can change the order of the rows in the data based on the values in a column by clicking the up/down arrow next to the column name.
- **Filter** You can filter to include only rows which have a certain value in a column by first clicking the small funnel icon labelled "Filter", then typing a desired value in the appropriate column.

Arrange and filter are known as data manipulation **verbs**. Individually, they describe a single simple manipulation of a dataset. It's surprising how many questions you can answer using just these two basic verbs:

- How old is the oldest dog in this data? To answer we can arrange the `animal_birth_month` column in increasing order and see Jack, a Pug from Queens, was born in January 1999 (this license was issued in May 2015, making Jack at least 16 at the time). You'll notice that there are other dogs with this same birthday.
- What range of license issue dates are in this data? Arrange `license_issued_date` once in increasing order and once in decreasing order, to find the issue dates range from 12th September 2015 to 31st December 2016.
- How many dogs licenses belong to dogs named Fido? Filter the `animal_name` column with `Fido`, and see "Showing … of 12 entries" - so 12!
- and many more…

While these verbs are powerful in their own right, their real power comes from combining them. For example, we can answer the more complicated question "Which dog named Fido is the oldest?" by first filtering then arranging.

The Viewer in RStudio, however, has two huge limitations:

- It's a point and click interface. This means to repeat the same operation again you need to remember exactly the steps of point, clicking and typing you performed to get to your answer. Consequently, it's hard to share those steps unambiguously with someone else, and it's hard to save your results for future.

- The manipulation verbs in the viewer are limited. There is no way to rearrange the columns, add new variables or calculate summaries like counts or averages.

You'll start this chapter by overcoming this first limitation. You won't use the Viewer to arrange and filter, you'll learn to write code to do the same operations. Then you'll increase your vocabulary of data manipulation verbs to include:

- selecting variables,
- adding variables,
- summarizing rows, and
- performing these operations on subsets of the data.

Combining these verbs you'll be able to answers questions like:

- How long are licences issued for?
- What are the most popular breeds?
- What names are most popular for licensed dogs in New York? Does this very geographically?
- When are dogs born?

FIXME: update these question to reflect things that are actually done in this and later chapters.

To master data manipulation you need to master two pieces:

- How to describe the action you want with the data manipulation verbs individually. This is a language specific skill - in this chapter, you'll use the functions in the dplyr package.
- Identifying which verbs, and in which order to apply them, to answer a question of interest. This skill will translate across all technologies, but it takes a little longer to master.

7.2.1 EXERCISE: POINT AND CLICK DATA MANIPULATION

Using the RStudio Viewer answer the following questions:

- How many dog licenses belong to dogs named "Queen" that live in "Queens"?

7.3 EXPLORING DATA IN THE CONSOLE

Let's take a look at the data in the console:

```
dog_licenses
```

```
## # A tibble: 118,542 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1        533 BONITA      F             2013-05-01 00:00:00 Unknown
##  2        548 ROCKY       M             2014-05-01 00:00:00 Labrador ~
##  3        622 BULLY       M             2010-07-01 00:00:00 American ~
##  4        633 COCO        M             2005-02-01 00:00:00 Labrador ~
##  5        655 SKI         F             2012-09-01 00:00:00 American ~
##  6        872 CHASE       M             2013-11-01 00:00:00 Shih Tzu
##  7        874 CHEWY       M             2014-09-01 00:00:00 Shih Tzu
##  8        875 CHASE       M             2008-08-01 00:00:00 Labrador ~
##  9        893 MILEY       F             2008-07-01 00:00:00 Boxer
## 10        919 KENZI       F             2010-05-01 00:00:00 Schnauzer~
## # ... with 118,532 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

Notice that in contrast to the Viewer you only see the first 10 rows of the dataset, and just the first few columns. The number of columns you see depends on the width of your console, so you may see more or fewer than displayed here. You should also note some of the contents of the columns have been abbreviated. The … at the end of some values in `breed_name` indicates these values have been truncated for display purposes.

You'll be using the dplyr package for data manipulation. Since it is part of the tidyverse, you'll need to load the tidyverse package to begin:

```
library(tidyverse)
```

7.3.1 RE-ARRANGING ROWS

You can reorder rows of data with the dplyr function `arrange()`. The arrange function takes a tibble as its first argument and column names

as the remaining arguments. The result will have the rows ordered in increasing value of the specified column. For example, to find the licenses belonging to the oldest dogs we arrange `dog_licenses` using the `animal_birth_month` column:

```
arrange(dog_licenses, animal_birth_month)
```

```
## # A tibble: 118,542 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1      15568 JACK        M             1999-01-01 00:00:00 Pug
##  2      23695 KATTY       F             1999-01-01 00:00:00 Chihuahua
##  3     101309 TOMMY       M             1999-01-01 00:00:00 Unknown
##  4       1598 SARAH       F             1999-01-01 00:00:00 West High~
##  5       8628 DOMINO      M             1999-01-01 00:00:00 Labrador ~
##  6      15733 BRINKS      M             1999-01-01 00:00:00 Yorkshire~
##  7      30419 LUCKY       M             1999-01-01 00:00:00 Unknown
##  8      31348 MAGGIE      F             1999-01-01 00:00:00 German Sh~
##  9      34685 COOKIE      M             1999-01-01 00:00:00 Pomeranian
## 10      37194 DARCY       F             1999-01-01 00:00:00 Cocker Sp~
## # ... with 118,532 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

You'll see Jack the Pug that lives in Queens, just like you did in the Viewer.

To arrange the rows by decreasing value, you need to wrap the column name in `desc()` (short for *desc*ending order). For instance to find the youngest dogs:

```
arrange(dog_licenses, desc(animal_birth_month))
```

```
## # A tibble: 118,542 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1     120352 MARLEY      M             2016-12-01 00:00:00 Cocker Sp~
##  2     121981 MR.         M             2016-12-01 00:00:00 Chihuahua~
##  3     120501 RORY        M             2016-12-01 00:00:00 Unknown
```

```
##  4     122028 TAQUITO     M             2016-12-01 00:00:00 Papillon
##  5     115820 REX         M             2016-11-01 00:00:00 Maltese
##  6     121727 CHANDERBAL~ M             2016-11-01 00:00:00 Havanese
##  7     115777 ANGEL       F             2016-11-01 00:00:00 Poodle, M~
##  8     118175 MASON       M             2016-11-01 00:00:00 American ~
##  9     121601 TEDDY       M             2016-11-01 00:00:00 Havanese
## 10     120995 LOLA        F             2016-11-01 00:00:00 Morkie
## # ... with 118,532 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

As another example, to find the earliest issue date we can order by increasing `license_issued_date`:

```
arrange(dog_licenses, license_issued_date)
```

```
## # A tibble: 118,542 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1          1 QUEEN       F             2013-04-01 00:00:00 Akita Cro~
##  2          2 CHEWBACCA   F             2012-06-01 00:00:00 Labrador ~
##  3          3 IAN         M             2006-01-01 00:00:00 Unknown
##  4          7 LOLA        F             2009-06-01 00:00:00 Maltese
##  5          4 PAIGE       F             2014-07-01 00:00:00 American ~
##  6          5 BUDDY       M             2008-04-01 00:00:00 Unknown
##  7          8 YOGI        M             2010-09-01 00:00:00 Boxer
##  8         10 MUNECA      F             2013-05-01 00:00:00 Beagle
##  9         27 BESS        F             2010-09-01 00:00:00 Beagle
## 10         26 BIGS        M             2004-12-01 00:00:00 American ~
## # ... with 118,532 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

The first row is the record with the earliest issue date, but we can't actually see that date because the column `license_issued_date` isn't being displayed due to space. One solution is to extract only the columns we are interested in, a manipulation known as selecting columns.

7.3.2 EXERCISE: ARRANGING CHARACTER STRINGS

Use `arrange()` to order the dog licenses by `animal_name` in increasing order. What does this tell you about the way R treats punctuation and numbers when dealing with alphabetical order?

7.4 HOW CAN I SELECT SUBSETS OF MY DATA?

Two verbs are used to subset data:

- `select()` to select columns
- `filter()` to select rows

You'll learn about these two functions in this section, along with learning about a way to chain together multiple operations on a dataset.

7.4.1 SELECTING COLUMNS

The `select()` function in dplyr is used to extract some subset of columns (but keep all the rows) from a tibble. Just like `arrange()`, it takes a tibble as its first argument and column names as the remaining arguments. For example, to keep only the `animal_name` column:

```
select(dog_licenses, animal_name)
```

```
## # A tibble: 118,542 x 1
##    animal_name
##    <chr>
##  1 BONITA
##  2 ROCKY
##  3 BULLY
##  4 COCO
##  5 SKI
##  6 CHASE
##  7 CHEWY
##  8 CHASE
##  9 MILEY
## 10 KENZI
## # ... with 118,532 more rows
```

You can provide additional column names as arguments to keep additional specified columns, for example to keep `animal_name` and `breed_name`:

```
select(dog_licenses, animal_name, breed_name)
```

```
## # A tibble: 118,542 x 2
##    animal_name breed_name
##    <chr>       <chr>
##  1 BONITA      Unknown
##  2 ROCKY       Labrador Retriever Crossbreed
##  3 BULLY       American Pit Bull Terrier/Pit Bull
##  4 COCO        Labrador Retriever
##  5 SKI         American Pit Bull Terrier/Pit Bull
##  6 CHASE       Shih Tzu
##  7 CHEWY       Shih Tzu
##  8 CHASE       Labrador Retriever
##  9 MILEY       Boxer
## 10 KENZI       Schnauzer, Miniature
## # ... with 118,532 more rows
```

To return to finding the earliest issue date, you need to first arrange by increasing `license_issued_date` and then select the `license_issued_date` column. One approach is to store the result of the arrange step,

```
dog_by_date <- arrange(dog_licenses, license_issued_date)
```

Then apply the select step to this object:

```
select(dog_by_date, license_issued_date)
```

```
## # A tibble: 118,542 x 1
##    license_issued_date
##    <date>
##  1 2014-09-12
##  2 2014-09-12
##  3 2014-09-12
##  4 2014-09-12
```

```
##  5 2014-09-12
##  6 2014-09-12
##  7 2014-09-12
##  8 2014-09-13
##  9 2014-09-13
## 10 2014-09-13
## # ... with 118,532 more rows
```

There are lots of shortcuts you can use with `select()` to avoid having to type out all the variables you want to keep. For example, you can ask for all the columns that start with a certain string:

```
select(dog_licenses, starts_with("Animal"))
```

```
## # A tibble: 118,542 x 3
##    animal_name animal_gender animal_birth_month
##    <chr>       <chr>         <dttm>
##  1 BONITA      F             2013-05-01 00:00:00
##  2 ROCKY       M             2014-05-01 00:00:00
##  3 BULLY       M             2010-07-01 00:00:00
##  4 COCO        M             2005-02-01 00:00:00
##  5 SKI         F             2012-09-01 00:00:00
##  6 CHASE       M             2013-11-01 00:00:00
##  7 CHEWY       M             2014-09-01 00:00:00
##  8 CHASE       M             2008-08-01 00:00:00
##  9 MILEY       F             2008-07-01 00:00:00
## 10 KENZI       F             2010-05-01 00:00:00
## # ... with 118,532 more rows
```

Take a look in the "Useful functions" section of the `select()` help page for a complete list:

```
?dplyr::select
```

7.4.2 EXERCISE: FIND THE LATEST LICENSE ISSUE DATE

Combine `arrange()` and `select()` to confirm the last issue date in this dataset is 31st December 2016.

7.4.3 COMBINING OPERATIONS WITH THE PIPE %>%

You've seen you can combine data manipulation steps to do more complicated tasks, but so far you've done so by saving an intermediate object, in our previous example the object `dog_by_date`:

```
dog_by_date <- arrange(dog_licenses, license_issued_date)
select(dog_by_date, license_issued_date)
```

The pipe, `%>%`, is an operator that allows you to chain together operations without intermediate objects and maintain readability. The name, "pipe", comes from the plumbing kind of pipe, not the smoking kind, and references the idea of objects flowing out of one function and into another. Let's just look at the first step in our manipulation:

```
arrange(dog_licenses, license_issued_date)
```

With the pipe this can be rewritten as:

```
dog_licenses %>% arrange(license_issued_date)
```

The pipe takes the object on the left hand side and passes it as the first argument to the function on the right hand side. So, here the `dog_licenses` dataset is passed to the first argument of `arrange()`. Inside `arrange()` we can then list any additional arguments as we normally would.

The pipe works very nicely with the data manipulation verbs because every verb expects a tibble as its first argument and returns a tibble. This means the result of one operation is easily piped into the next operation, allowing you to chain together multiple steps. For instance, piping the result of the arrange step above into the `select()` function:

```
dog_licenses %>%
  arrange(license_issued_date) %>%
  select(license_issued_date)
```

When you see the pipe, read it as "and then". So, the above code would be read:

> Take the dog_licenses data, **and then** arrange the rows by the `license_issued_date`, **and then** select the column `license_issued_date`.

The result is code that matches very closely how we might describe the steps we performed in natural language. It's so natural that for the remainder of the chapter we'll use the pipe when combining data manipulation steps.

7.4.4 EXERCISE: READING ALOUD

Read the following code aloud to your neighbor (or cat, dog, or rubber duck). Remember to pronounce `%>%` as "and then".

```
dog_licenses %>%
  arrange(license_issued_date) %>%
  select(license_expired_date)
```

What question might it answer?

7.4.5 EXERCISE: USING THE PIPE

Re-write this snippet of code to use the pipe:

```
select(dog_licenses, animal_name, breed_name)
```

Use the pipe to re-write this snippet of code to avoid the intermediate variable:

```
name_and_breed <- select(dog_licenses, animal_name, breed_name)
arrange(name_and_breed, breed_name)
```

7.4.6 FILTERING TO KEEP A SUBSET OF ROWS

The function to filter rows of a tibble is `filter()`. Like `arrange()` and `select()`, its first argument is a tibble. The remaining arguments describe which rows to **keep**. The rows to keep are specified with a

logical expression - something that is either `TRUE` or `FALSE`. The rows where this expression is `TRUE` will be returned.

One of the simplest kinds of logical expression is a test for equality with the `==` operator. For example, to keep the rows where `animal_name` is `BRUNO` you could do:

```
dog_licenses %>% filter(animal_name == "BRUNO")
```

```
## # A tibble: 272 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1      12001 BRUNO       M             2010-05-01 00:00:00 American ~
##  2      27228 BRUNO       M             2013-07-01 00:00:00 Jack Russ~
##  3      68192 BRUNO       M             2002-01-01 00:00:00 Shih Tzu
##  4      70175 BRUNO       M             2015-12-01 00:00:00 Chihuahua~
##  5     120562 BRUNO       M             2016-05-01 00:00:00 Labrador ~
##  6       3915 BRUNO       M             2014-03-01 00:00:00 Doberman ~
##  7       9614 BRUNO       M             2014-12-01 00:00:00 Boxer
##  8      15606 BRUNO       M             2015-02-01 00:00:00 French Bu~
##  9      32742 BRUNO       M             2014-03-01 00:00:00 Maltipoo
## 10      34299 BRUNO       M             2003-01-01 00:00:00 Unknown
## # ... with 262 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

You could read this code aloud as:

> Take the dog_licenses data, **and then** filter for only rows when `animal_name` is equal to "BRUNO"

The logical expression, `animal_name == "BRUNO"`, will be `TRUE` when the value of the `animal_name` column is exactly equal to the character string `"BRUNO"` - any differences in characters, case, or whitespace will result in `FALSE`.

Above, each value within a column was compared against the same fixed string (`"BRUNO"`). You can also compare the values from two columns against each other by including a column name on each side of the `==`

sign. In programming terms this is known as element-wise comparison. For example, `license_issued_date == animal_birth_month` will return `TRUE` for a row only if for that row the date the license was issued is the exact same date as the birth month for the dog. If you take a look:

```
dog_licenses %>%
  filter(license_issued_date == animal_birth_month)
```

```
## # A tibble: 0 x 15
## # ... with 15 variables: row_number <dbl>, animal_name <chr>,
## #   animal_gender <chr>, animal_birth_month <dttm>, breed_name <chr>,
## #   borough <chr>, zip_code <dbl>, community_district <dbl>,
## #   census_tract_2010 <dbl>, neighborhood_tabulation_area <chr>,
## #   city_council_district <dbl>, congressional_district <dbl>,
## #   state_senatorial_district <dbl>, license_issued_date <date>,
## #   license_expired_date <date>
```

There is no output (apart from the column names), which means that no rows satisfy this criteria.

Remember that a string is surrounded by quotes while a column name is not. When you are reading code, look for the quotes to figure out if the comparison is to a string, or (by the absence of quotes) to the strings within a column. Use the same strategy to figure out where the quotes should be in your own code, but beware: misplacing quotes often won't result in an error, but instead a result that you weren't expecting. For example, I might be interested in the licenses issued to male dogs and try:

```
dog_licenses %>%
  filter("animal_gender" == "M")
```

```
## # A tibble: 0 x 15
## # ... with 15 variables: row_number <dbl>, animal_name <chr>,
## #   animal_gender <chr>, animal_birth_month <dttm>, breed_name <chr>,
## #   borough <chr>, zip_code <dbl>, community_district <dbl>,
## #   census_tract_2010 <dbl>, neighborhood_tabulation_area <chr>,
## #   city_council_district <dbl>, congressional_district <dbl>,
## #   state_senatorial_district <dbl>, license_issued_date <date>,
## #   license_expired_date <date>
```

The result has zero rows, which would suggest there are no such licenses, but in fact this is the answer to a different question. Can you see what is wrong with the code? By surrounding `animal_gender` in quotes, R has interpreted the comparison as: is the string `"animal_gender"` equal to the string `"M"`?. The answer is `FALSE`, and no rows are returned. I actually wanted to compare the `animal_gender` column to the string `"M"`, so `animal_gender` should have no quotes around it:

```
dog_licenses %>%
  filter(animal_gender == "M")
```

```
## # A tibble: 64,770 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1        548 ROCKY       M             2014-05-01 00:00:00 Labrador ~
##  2        622 BULLY       M             2010-07-01 00:00:00 American ~
##  3        633 COCO        M             2005-02-01 00:00:00 Labrador ~
##  4        872 CHASE       M             2013-11-01 00:00:00 Shih Tzu
##  5        874 CHEWY       M             2014-09-01 00:00:00 Shih Tzu
##  6        875 CHASE       M             2008-08-01 00:00:00 Labrador ~
##  7        976 APOLLO      M             2014-10-01 00:00:00 American ~
##  8       1297 JERRY       M             2009-06-01 00:00:00 Labrador ~
##  9       2133 SIMON       M             2010-12-01 00:00:00 Havanese
## 10       2289 BUDDY       M             2012-06-01 00:00:00 Labrador ~
## # ... with 64,760 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

The operator, `==` (you can read as "is equal to", or simply "equals"), is a specific kind of comparison. Other comparisons include:

Operator	Meaning
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to

7.4.7 EXERCISE: ENTERPRISING DOGS

Are there any dogs called "Spock", "Picard", or "Janeway"?

These are Star Trek characters, can you find any dogs with a name from one of your favorite books, TV shows, or movies?

7.4.8 EXERCISE: EXPIRED LICENSES

This code creates a variable that contains the date for the start of the year 2016:

```
start_of_2016 <- as.Date("2016-01-01")
```

Use `filter()` with this variable to find:

- The dog licenses that were issued before 2016
- The dog licenses that expire before 2016

7.4.9 MORE COMPLICATED EXPRESSIONS

Logical expressions can be combined with logical operators to construct more complicated expressions. For example, the AND operator, `&`, returns `TRUE` only if the expressions on both sides of it are `TRUE`. For example, you've seen, you can find the licenses to dogs called `"BRUNO"`:

```
dog_licenses %>%
  filter(animal_name == "BRUNO")
```

```
## # A tibble: 272 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1      12001 BRUNO       M             2010-05-01 00:00:00 American ~
##  2      27228 BRUNO       M             2013-07-01 00:00:00 Jack Russ~
##  3      68192 BRUNO       M             2002-01-01 00:00:00 Shih Tzu
##  4      70175 BRUNO       M             2015-12-01 00:00:00 Chihuahua~
##  5     120562 BRUNO       M             2016-05-01 00:00:00 Labrador ~
##  6       3915 BRUNO       M             2014-03-01 00:00:00 Doberman ~
##  7       9614 BRUNO       M             2014-12-01 00:00:00 Boxer
```

```
##  8      15606 BRUNO       M             2015-02-01 00:00:00 French Bu~
##  9      32742 BRUNO       M             2014-03-01 00:00:00 Maltipoo
## 10      34299 BRUNO       M             2003-01-01 00:00:00 Unknown
## # ... with 262 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

And you could find the dog licenses issued to dogs that live in Brooklyn:

```
dog_licenses %>%
  filter(borough == "Brooklyn")
```

```
## # A tibble: 29,334 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1       2895 FUDGE       M             2014-07-01 00:00:00 American ~
##  2       4057 STAR        F             2011-01-01 00:00:00 Poodle
##  3      74463 MUNECA      F             2011-09-01 00:00:00 Chihuahua~
##  4      76232 KATTY       F             2002-06-01 00:00:00 Chihuahua
##  5      85113 SHADOW      M             2015-03-01 00:00:00 American ~
##  6      85997 SPARKIE     M             2013-08-01 00:00:00 Maltese C~
##  7      92451 SNOW        M             2014-07-01 00:00:00 Maltese
##  8     104256 BELLA       F             2016-07-01 00:00:00 Maltese
##  9      10389 SPARKLE     F             2006-01-01 00:00:00 Schnauzer~
## 10      82492 UNKNOWN     F             2015-11-01 00:00:00 Pomeranian
## # ... with 29,324 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

If you want to find the licenses that are to dogs named Bruno in Brooklyn, you could combine the two logical statements with `&`:

```
dog_licenses %>%
  filter((animal_name == "BRUNO") & (borough == "Brooklyn"))
```

```
## # A tibble: 55 x 15
```

```
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1       8496 BRUNO       M             2005-04-01 00:00:00 American ~
##  2      65466 BRUNO       M             2012-11-01 00:00:00 Golden Re~
##  3     115999 BRUNO       M             2009-01-01 00:00:00 Pug
##  4     118963 BRUNO       M             2005-01-01 00:00:00 Unknown
##  5      10505 BRUNO       M             2006-04-01 00:00:00 Bull Dog,~
##  6      14444 BRUNO       M             2007-01-01 00:00:00 Cocker Sp~
##  7      14690 BRUNO       M             2014-04-01 00:00:00 Shih Tzu
##  8      47454 BRUNO       M             2011-06-01 00:00:00 Pug
##  9      58918 BRUNO       M             2006-04-01 00:00:00 Bull Dog,~
## 10     105964 BRUNO       M             2016-04-01 00:00:00 Bull Dog,~
## # ... with 45 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

The parentheses around each logical expression are optional, but can help visually to group the components to `&`, especially if those logical expressions get more complicated.

If you want to create an "or" type expression, like "licenses to dogs named BRUNO **or** dogs named BRUCE", you need to combine two comparisons with the OR operator, `|`.

```
dog_licenses %>%
  filter((animal_name == "BRUNO") | (animal_name == "BRUCE"))
```

If you find yourself combining lots of comparisons on the same column with `|`, like dogs named `BRUNO`, `BRUCE` or `BRADY`:

```
dog_licenses %>%
  filter((animal_name == "BRUNO") | (animal_name == "BRUCE") | (animal_name == "BRADY"))
```

You can save a lot of typing with `%in%`:

```
dog_licenses %>%
  filter(animal_name %in% c("BRUNO", "BRUCE", "BRADY"))
```

On the right hand side of `%in%` the function `c()`, **c**ombines many single values into a vector. `%in%` will return `TRUE` for an element of the left hand side if it is contained in the vector on the right hand side.

7.4.10 EXERCISE: EXPIRED LICENSES

This code creates two variables that contain the dates for the start and end of the year 2016:

```
start_of_2016 <- as.Date("2016-01-01")
endt_of_2016 <- as.Date("2016-12-31")
```

Use `filter()` with these variables to find the dog licenses that expire during 2016.

7.5 HOW CAN I CALCULATE NEW VALUES?

So far, you've been manipulating the columns that already exist in a dataset, but what if you want to add new ones? The function `mutate()` handles this kind of operation.

To see how this works let's start with a logical expression: `animal_name == "CHASE"`. If you used this with `filter()`, you would get all the rows back where the license was issued to a dog named `CHASE`. Let's say instead of subsetting the data, you want to add a column called `called_chase` that contained the `TRUE` and `FALSE` result. You might do this for example, if you are interested in comparing the two groups of dogs, rather than just keeping one of them. With `mutate()` after passing in the data, you pass named arguments, where the name is the name you desire for the new column, and its value is the way to calculate it:

```
dog_licenses %>%
  mutate(called_chase = animal_name == "CHASE") %>%
  select(animal_name, called_chase)
```

```
## # A tibble: 118,542 x 2
##    animal_name called_chase
##    <chr>       <lgl>
##  1 BONITA      FALSE
##  2 ROCKY       FALSE
##  3 BULLY       FALSE
##  4 COCO        FALSE
##  5 SKI         FALSE
##  6 CHASE       TRUE
##  7 CHEWY       FALSE
```

```
##  8 CHASE       TRUE
##  9 MILEY       FALSE
## 10 KENZI       FALSE
## # ... with 118,532 more rows
```

> Take `dog_licences`, **and then**, mutate to add a column called `called_chase` which is the result of testing whether `animal_name` is exactly `"CHASE"`, **and then**, select the columns `animal_name` and `called_chase`.

The select statement isn't crucial to the calculation here, but it does help me draw your attention to the columns that were involved in this step.

This can be a useful intermediate step in filtering, since it gives you a chance to examine the logical statement before using it to filter:

```
dog_licenses %>%
  mutate(called_chase = animal_name == "CHASE") %>%
  filter(called_chase)
```

```
## # A tibble: 126 x 16
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>              <chr>
##  1        872 CHASE       M             2013-11-01 00:00:00 Shih Tzu
##  2        875 CHASE       M             2008-08-01 00:00:00 Labrador ~
##  3      32652 CHASE       M             2013-09-01 00:00:00 Yorkshire~
##  4      42125 CHASE       M             2015-08-01 00:00:00 Chihuahua
##  5     109847 CHASE       M             2013-04-01 00:00:00 Terrier m~
##  6     114557 CHASE       M             2009-07-01 00:00:00 Siberian ~
##  7      33434 CHASE       M             2014-08-01 00:00:00 Schnauzer~
##  8      45142 CHASE       M             2005-01-01 00:00:00 Unknown
##  9       2528 CHASE       M             2011-10-01 00:00:00 Lhasa Apso
## 10       3426 CHASE       M             2014-03-01 00:00:00 Yorkshire~
## # ... with 116 more rows, and 11 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>,
## #   called_chase <lgl>
```

Take a closer look at the argument to mutate:

```
called_chase = animal_name == "CHASE"
```

On the left of the `=` is the argument name, `called_chase`. This is a name you choose - it should be descriptive and follow good style. On the right of the `=` is an expression that must either return as many values as there are rows, or a single value. Here, the logical expression involves the column `animal_name` so it returns as many values as there are rows.

If you would prefer the values to be something other than `TRUE` or `FALSE`, instead maybe you want them to be something like `"called chase"` or `"not called chase"`, we would need to use the `ifelse()`.

A call to `ifelse()` takes the form:

```
ifelse(test, yes, no)
```

Where `test` is a logical expression, `yes` the value for the elements that return `TRUE`, and `no` the value for the elements that return `FALSE`. (Both `yes` and `no` could be other column names, in which case, the corresponding element of `yes` would be returned for `TRUE` elements).

```
dog_licenses %>%
  mutate(
    called_chase = ifelse(animal_name == "CHASE", "called chase", "not called chase")
    ) %>%
  select(animal_name, called_chase)
```

```
## # A tibble: 118,542 x 2
##    animal_name called_chase
##    <chr>       <chr>
##  1 BONITA      not called chase
##  2 ROCKY       not called chase
##  3 BULLY       not called chase
##  4 COCO        not called chase
##  5 SKI         not called chase
##  6 CHASE       called chase
##  7 CHEWY       not called chase
##  8 CHASE       called chase
##  9 MILEY       not called chase
## 10 KENZI       not called chase
## # ... with 118,532 more rows
```

FIXME: add diagram showing how ifelse works

You can perform multiple mutate steps at once by passing more arguments to `mutate()`, so an alternative way of writing the above code (with more keystrokes, but with lines that are shorter) would be:

```
dog_licenses %>%
  mutate(
    is_chase = animal_name == "CHASE",
    called_chase = ifelse(is_chase, "called chase", "not called chase")
    ) %>%
  select(animal_name, is_chase, called_chase)
```

```
## # A tibble: 118,542 x 3
##    animal_name is_chase called_chase
##    <chr>       <lgl>    <chr>
##  1 BONITA      FALSE    not called chase
##  2 ROCKY       FALSE    not called chase
##  3 BULLY       FALSE    not called chase
##  4 COCO        FALSE    not called chase
##  5 SKI         FALSE    not called chase
##  6 CHASE       TRUE     called chase
##  7 CHEWY       FALSE    not called chase
##  8 CHASE       TRUE     called chase
##  9 MILEY       FALSE    not called chase
## 10 KENZI       FALSE    not called chase
## # ... with 118,532 more rows
```

Notice that the computation for the `called_chase` column refers to the `is_chase` column. The arguments to mutate are computed in order, so columns created later in the same `mutate()` can refer to columns created earlier.

Arithmetic is another common operation that returns as many elements as there are rows. For instance we could see how long licenses are issued for:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  select(license_duration)
```

```
## # A tibble: 118,542 x 1
```

```
##    license_duration
##    <drtn>
##  1 1118 days
##  2 1826 days
##  3  697 days
##  4 1096 days
##  5 1826 days
##  6  731 days
##  7  731 days
##  8 1097 days
##  9  421 days
## 10  402 days
## # ... with 118,532 more rows
```

The output shows us licenses aren't issued for a standard time period. In these first rows, there are some licenses issued for a whole number of years: 2 (731 days), 3 (1097 days) and 5 (1826 days). However, others seem to be for fractions of years like 421 days.

FIXME: provide link to RStudio data mini cheatsheet with other functions that are useful with mutate.

You can use `mutate()` with operations that give one number based on all the rows:

```
dog_licenses %>%
  mutate(
    license_duration = license_expired_date - license_issued_date,
    avg_duration = mean(license_duration)) %>%
  select(license_duration, avg_duration)
```

```
## # A tibble: 118,542 x 2
##    license_duration avg_duration
##    <drtn>           <drtn>
##  1 1118 days        467.321 days
##  2 1826 days        467.321 days
##  3  697 days        467.321 days
##  4 1096 days        467.321 days
##  5 1826 days        467.321 days
##  6  731 days        467.321 days
##  7  731 days        467.321 days
##  8 1097 days        467.321 days
```

```
##  9  421 days        467.321 days
## 10  402 days        467.321 days
## # ... with 118,532 more rows
```

You'll get back the original number of rows, but the single value will be repeated in all of them - on average licenses are issued for 467.321 days. You'll see a different verb, `summarise()` that collapses many rows into one later in this chapter.

7.5.1 EXERCISE: AGES OF DOGS

Use `mutate()` to add a column `age_at_issue` that contains the dogs approximate age on the day the license was issued.

7.5.2 EXERCISE: UNKNOWN BREEDS

Use `mutate()` along with `ifelse()` to create a column `breed` that takes values `"unknown"` if `breed_name` is `"unknown"` and `"known"` otherwise.

Extra challenge Can you figure out if the licenses issued to unknown breed dogs are of longer or shorter duration on average than known breed dogs?

7.5.3 EXERCISE: NAME LENGTH

The function `str_length()` in the stringr package finds the length of character strings. Replace the `___` in following code to add a column called `name_length` that contains the length of the dog's name:

```
dog_licenses %>%
  ___(___ = stringr::str_length(animal_name))
```

Now add an `arrange()` step to find the licenses issued to dogs with the longest names?

7.6 HOW CAN I TELL WHAT'S GONE WRONG IN MY PROGRAMS?

FIXME: The errors shown in the markdown are way more informative than those in the Console. Try to get the error displaying in the book like they do for someone in the console?

Let me share an interaction my (CVW's) husband had at a Trader Joes (a small specialized supermarket) soon after we arrived in the USA from New Zealand.

Josh: "Do you sell *bat-trees*?"

Store-person: "What?"

Josh: "Do you sell *bat-trees*?"

Store person: "Huh? *Bat…trees*?"

Josh: "Do you sell *bat-er-ries*?

Store-person: "Oh…you mean batteries! No. We don't sell batteries."

This is a pretty accurate analogy for what it feels like when you are learning R. You know what you want, but you have to ask for it in a way R understands. When R doesn't understand you, or when R can't give you what you want, you'll get an error. You'll know when you get one in R because the only output you see will start with `Error`.

```
buy_batteries(store = "Trader Joes")
```

```
## Error in buy_batteries(store = "Trader Joes"): could not find function "buy_batterie
```

It's also a good illustration of the two kinds of problems that occur: syntax errors and runtime errors. Syntax errors are like the "What? Huh?" moments. R doesn't understand what you are asking it to do, because something about the way you are asking doesn't conform to what R expects and it will not even try to run your code. Runtime errors are more like the "No, I can't help you" moments. R understands what you are asking and runs your code, but during the run something goes wrong and R has to stop running the code.

You generally want to make sure you've ruled out syntax errors before assuming it's a runtime error. Unfortunately, R doesn't distinguish these two types of error in its output, so we'll discuss some of the most common examples in the following sections.

This analogy is also a reminder that sometimes you'll have to repeat yourself. With R, giving the exact same instruction should always result in the exact same error, but it's not uncommon, even for longtime R users, to run and edit a line of code multiple times before it runs without error.

There is one flaw in this analogy: R isn't a real person. So, if you need to vent *your* frustration by cursing it, insulting its parentage or storming off, it's fine, no one's feelings will be hurt. Of course, it won't change R's response…

7.6.1 COMMON SYNTAX ERRORS

Syntax errors occur when your code can't be broken into its component pieces by R. For example, R expects the arguments to a function to start after the opening parenthesis (`(`), be separated by a comma (`,`) and finish at the closing parenthesis (`)`). When R sees this code:

```
filter(dog_licenses breed_name == "Finnish Lapphund")
```

```
## Error: <text>:1:21: unexpected symbol
## 1: filter(dog_licenses breed_name
##                         ^
```

The missing comma means R can't figure out where the first argument to `filter()` ends: is it after `dog_licenses`, after `breed_name`, or after `==`?

Take a closer look at the error message. Error messages always begin with `Error`, then optionally the name of the function that returned an error (not in this example), followed by a `:`, and some description of the error that occurred. The message `unexpected symbol` is one common kind of syntax error - in this case R encountered some code where it was expecting a comma or closing parenthesis. We can fix it by putting in the missing comma:

```
filter(dog_licenses, breed_name == "Finnish Lapphund")
```

```
## # A tibble: 1 x 15
##   row_number animal_name animal_gender animal_birth_month  breed_name
##        <dbl> <chr>       <chr>         <dttm>              <chr>
## 1     118408 FREDDIE     M             2011-11-01 00:00:00 Finnish L~
## # ... with 10 more variables: borough <chr>, zip_code <dbl>,
## #   community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

Syntax errors are usually the result of typos. Some things to keep an eye out for:

- If you are modelling your code on an example, pay very close attention to the punctuation: commas `,`, parenthesis `(`, `)`, brackets, `[`, `]`, and quotes `"`, `'`. Every opening parenthesis, bracket, or quote needs a matching closing one, and they must be closed in the reverse order they were opened.
- New lines don't matter if they happen between arguments, or after pipe operators, but they can be problematic in other locations.
- R ignores other whitespace (spaces or tabs) unless it's inside a character string (i.e. inside quotes), so different spacing shouldn't be the cause of an error, but it is a good idea to follow good style for spacing. FIXME: link to style guide section.

7.6.2 EXERCISE: SYNTAX ERRORS

Fix these **syntax** errors.

- ```
  dog_licenses %>%
    arrange(desc(license_expired_date)))
  ```

  ```
  ## Error: <text>:2:38: unexpected ')'
  ## 1: dog_licenses %>%
  ## 2:   arrange(desc(license_expired_date)))
  ##                                         ^
  ```

- ```
  dog_licenses %>%
    mutate(
      month_born = lubridate::month(animal_birth_month)
      year_born = lubridate::year(animal_birth_month))
  ```

  ```
  ## Error: <text>:4:5: unexpected symbol
  ## 3:     month_born = lubridate::month(animal_birth_month)
  ## 4:     year_born
  ##        ^
  ```

- ```
  dog_licenses %>%
    filter(animal_name == "BRUNO)
  ```

  ```
  ## Error: <text>:2:25: unexpected INCOMPLETE_STRING
  ## 1: dog_licenses %>%
  ## 2:   filter(animal_name == "BRUNO)
  ##                            ^
  ```

  (If you run this code in the Console, you might not get an error, but you should see a `+` on a new line, a signal that R is waiting for more input and a clue that there is something missing in this code).

- ```
  dog_licenses
    %>% filter(animal_gender == "M")
  ```

  ```
  ## Error: <text>:2:3: unexpected SPECIAL
  ## 1: dog_licenses
  ## 2:   %>%
  ##      ^
  ```

7.6.3 COMMON RUNTIME ERRORS

Runtime errors come in an infinite number of flavors because there are so many ways that you ask for that might be impossible to do. For example you might try to do arithmetic with character strings:

```
"apple" + "banana"
```

```
## Error in "apple" + "banana": non-numeric argument to binary operator
```

Or try to filter with something that isn't a logical:

```
dog_licenses %>% filter(animal_name)
```

```
## Error: Argument 2 filter condition does not evaluate to a logical vector
```

Perhaps the most common runtime error is of the form `Error: object not found`:

```
an_object_i_dont_have
```

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

This is R complaining that you've asked it to operate on an object that it doesn't know about. Often this is actually a typo in disguise, for example you've misspelled the name of the object,

```
my_object <- 12
my_objet
```

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

you've used the wrong case,

```
My_object
```

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

or you've forgotten that you've used a separator

```
myobject
```

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

This error also often arises when you forgot quotes around strings. For example, if we want all the dogs that are male, and try

```
dog_licenses %>% filter(animal_gender == M)
```

```
## Error: object 'M' not found
```

you get an error because R is looking for an object called `M` to compare to the values in the column called `animal_gender`. What we really wanted to do was compare the values in `animal_gender` to the string `"M"`:

```
dog_licenses %>% filter(animal_gender == "M")
```

```
## # A tibble: 64,770 x 15
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1        548 ROCKY       M             2014-05-01 00:00:00 Labrador ~
##  2        622 BULLY       M             2010-07-01 00:00:00 American ~
##  3        633 COCO        M             2005-02-01 00:00:00 Labrador ~
##  4        872 CHASE       M             2013-11-01 00:00:00 Shih Tzu
##  5        874 CHEWY       M             2014-09-01 00:00:00 Shih Tzu
##  6        875 CHASE       M             2008-08-01 00:00:00 Labrador ~
##  7        976 APOLLO      M             2014-10-01 00:00:00 American ~
##  8       1297 JERRY       M             2009-06-01 00:00:00 Labrador ~
##  9       2133 SIMON       M             2010-12-01 00:00:00 Havanese
## 10       2289 BUDDY       M             2012-06-01 00:00:00 Labrador ~
## # ... with 64,760 more rows, and 10 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>
```

7.6.4 EXERCISE: OBJECT NOT FOUND

Fix these `object not found` errors. (Hint: the names of the objects or variables being created should give you a clue to the intent of the code)

- ```
  dog_licenses %>%
    mutate(year_issued = lubridate::year(Liscenceissuedate))
  ```

  ```
  ## Error in lubridate::year(Liscenceissuedate): object 'Liscenceissuedate' not found
  ```

- ```
  dogs_named_bruno <- dog_licenses %>%
    filter(animal_name == BRUNO)
  ```

  ```
  ## Error: object 'BRUNO' not found
  ```

7.6.5 WARNINGS AND MESSAGES

There are two other kinds of alerts R can give: warnings and messages. These can both appear in the console with the same color as an error,

but they are informational as opposed to fatal.

Messages are purely informational, for example when you read data in with `read_csv()` you get a message that describes the columns and their data types as parsed by the function:

```
sites <- read_csv("site.csv")
```

```
## Parsed with column specification:
## cols(
##   site_id = col_character(),
##   latitude = col_double(),
##   longitude = col_double()
## )
```

Warnings generally alert you that something was slightly unexpected but that R recovered and gave you a result anyway. Poorly formatted CSV files will often result in warnings from `read_csv()`:

```
bad_csv <- "
id,
1, 2
2, 1
3, 5
"
bad <- read_csv(bad_csv)
```

```
## Warning: Missing column names filled in: 'X2' [2]
```

Here there was a missing column name. `read_csv()` still returns an object but the warning alerts you that it made some assumption to get it, i.e. that it made up a column name:

```
bad
```

```
## # A tibble: 3 x 2
##      id    X2
##   <dbl> <dbl>
## 1     1     2
## 2     2     1
## 3     3     5
```

Warnings don't **stop** you from proceeding, but they should alert you to question whether you **should be** proceeding.

7.6.6 WHAT DO I DO WHEN I GET AN ERROR I CAN'T FIX?

- Check that the error is reproducible. Restart R with a clean slate and re-run your code up to and including the code that gives the error. This is the R version of the classic tech advice to "turn it off, then turn it on again". FIXME: link to reproducibility chapter.
- Try searching for it online: for example, searching for "R unexpected string constant" lead me to the question "Error: unexpected symbol/input/string constant/numeric constant/SPECIAL in my code" on StackOverflow which gives some great examples of ways this error might arise.
- Ask for help. You are most likely to get help when you can provide a reproducible example. Stack Overflow has detailed instruction on how to create a minimal reproducible example when asking a question to increase the chances that the question receives a specific and helpful answer. The key principles listed on the website recommends that an answer follows these guidelines:
    - Minimal – Use as little code as possible that still produces the same problem
    - Complete – Provide all parts someone else needs to reproduce your problem in the question itself
    - Reproducible – Test the code you're about to provide to make sure it reproduces the problem

7.7 HOW CAN I OPERATE ON SUBSETS OF MY DATA?

The syntax for `summarise()` is the same as `mutate()` but it expects operations that reduce all rows down to one row. Recall from `mutate()` that this code added the average license duration to every row of the data:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  mutate(avg_duration = mean(license_duration))
```

You actually saw this in one `mutate()` statement, but I've separated out the line that calculates the average so it's easier to see the difference with `summarise()`. See what happens when you switch out the final `mutate()` with `summarise()`:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  summarise(avg_duration = mean(license_duration))
```

```
## # A tibble: 1 x 1
##   avg_duration
##   <drtn>
## 1 467.321 days
```

Instead of the the one value repeated on every row, we get a new tibble with only one row, and a single column that corresponds to our requested summary.

Any function that takes many values and reduces them to one is a good candidate for `summarise()`, for example we could find the shortest licence duration by swapping in `min()` instead of `mean()`:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  summarise(shortest_duration = min(license_duration))
```

```
## # A tibble: 1 x 1
##   shortest_duration
##   <drtn>
## 1 1 days
```

Like `mutate()` you can also create multiple summary columns at once:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration)
  )
```

```
## # A tibble: 1 x 3
##   avg_duration shortest_duration longest_duration
##   <drtn>       <drtn>            <drtn>
## 1 467.321 days 1 days            2191 days
```

FIXME: link to cheatsheet with list of other useful functions.

Lot's of statistical operations produce one numbers summaries and are appropriate for use with `summarise()`: `sd()`, `min()`, `max()`, `mean()`, `median()`, `quantile()` (with a single argument). Whenever you are summarizing many rows, it's a good idea to keep track of how many rows were summarized. This is so common, dplyr provides a special function, `n()`, that simply counts the number of rows. To add it to your summary:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

```
## # A tibble: 1 x 4
##   avg_duration shortest_duration longest_duration n_licenses
##   <drtn>       <drtn>            <drtn>                <int>
## 1 467.321 days 1 days            2191 days            118542
```

Now, imagine you want this summary just for licenses issued to dogs in the Bronx. You might do something like:

> Take the `dog_licenses` data, **and then**, mutate to add a column called `license_duration`, **and then** filter to keep rows where the `borough` is `"Bronx"`, **and then** summarise to find the mean, min and max duration along with the number of rows.

In code:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  filter(borough == "Bronx") %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

```
## # A tibble: 1 x 4
##   avg_duration  shortest_duration longest_duration n_licenses
##   <drtn>        <drtn>            <drtn>                <int>
## 1 435.9884 days 2 days            1919 days             12043
```

But how does this compare to Brooklyn? You could do the same operation again, but now for Brooklyn:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  filter(borough == "Brooklyn") %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

```
## # A tibble: 1 x 4
##   avg_duration  shortest_duration longest_duration n_licenses
##   <drtn>        <drtn>            <drtn>                <int>
## 1 465.8845 days 2 days            2191 days             29334
```

What about Queens? This kind of operation—summarising different subsets of the same data—is so common there is a much easier way to do it: combining `summarise()` with `group_by()`.

The only difference in the code, is that instead of filtering for a specific `borough` we'll `group_by()` the column `borough`.

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  group_by(borough) %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

```
## # A tibble: 57 x 5
##    borough      avg_duration  shortest_duration longest_duration n_licenses
##    <chr>        <drtn>        <drtn>            <drtn>                 <int>
##  1 ARVERNE      334.0000 days 334 days           334 days                  1
##  2 Astoria      498.0000 days 239 days           757 days                  2
##  3 ASTORIA      387.6667 days 366 days           405 days                  3
##  4 B            347.0000 days 347 days           347 days                  1
##  5 Bayside      410.0000 days 410 days           410 days                  1
##  6 BELLE HARBOR 309.0000 days 309 days           309 days                  1
##  7 Briarwood    540.5000 days 358 days           723 days                  2
##  8 Bronx        435.9884 days   2 days          1919 days              12043
##  9 BRONX        370.3333 days  72 days           418 days                102
## 10 Brooklyn     465.8845 days   2 days          2191 days              29334
## # ... with 47 more rows
```

The `group_by()` verb doesn't perform any changes to the data except to add a signal that this data is now grouped. Subsequent operations will then happen within these groups. In the case of `summarise()` we now get one row per group, and these are all stacked together in our result.

You might have been a little surprised by the result above. I thought there were only five boroughs in New York (at least that's what the Beastie Boys told me). Notice some boroughs are represented more than once by variations in case or spelling: `Bronx`, `BRONX`. As far as `group_by()` is concerned these are distinct values of this variable. There also seem to be smaller designations than Borough in this data. You'll get a chance to try and resolve this in an exercise below.

7.7.1 EXERCISE: DOG BIRTH MONTHS

The following code creates a new column `month_born` that holds the name of the month the licensed dog was born:

```
dog_licenses %>%
  mutate(month_born = lubridate::month(animal_birth_month, label = TRUE))
```

Use a `group_by()` step and a `summarise()` step to find the number of dogs born in each month. Which month stands out? Can you guess why?

7.7.2 EXERCISE: ORDER MATTERS?

In the example above for dogs licensed in Brooklyn:

```
dog_licenses %>%
  mutate(license_duration = license_expired_date - license_issued_date) %>%
  filter(borough == "Brooklyn") %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

the `filter()` step came after the `mutate()` step. Does this matter?

- Swap the order in the code and see if you get the same results.
- Write out how you might describe the steps. Is it obvious you can swap the filter and mutate step and get the same results?
- Which steps can't you swap the order of? Why?
- *Despite giving the same results, some orderings of the data manipulation steps will take longer to compute. Can you guess why?*

7.7.3 EXERCISE: THE FIVE BOROUGHS

The column `neighborhood_tabulation_area` is a code for the "Neighborhood Tabulation Areas", and has been geo-coded from the licensee's

address (as opposed to self reported). The first two characters correspond to the Borough.

This code creates a new variable called `borough_code` that contains just these two characters:

```
dog_licenses %>%
  mutate(borough_code = stringr::str_sub(neighborhood_tabulation_area, 1, 2))
```

```
## # A tibble: 118,542 x 16
##    row_number animal_name animal_gender animal_birth_month  breed_name
##         <dbl> <chr>       <chr>         <dttm>               <chr>
##  1        533 BONITA      F             2013-05-01 00:00:00 Unknown
##  2        548 ROCKY       M             2014-05-01 00:00:00 Labrador ~
##  3        622 BULLY       M             2010-07-01 00:00:00 American ~
##  4        633 COCO        M             2005-02-01 00:00:00 Labrador ~
##  5        655 SKI         F             2012-09-01 00:00:00 American ~
##  6        872 CHASE       M             2013-11-01 00:00:00 Shih Tzu
##  7        874 CHEWY       M             2014-09-01 00:00:00 Shih Tzu
##  8        875 CHASE       M             2008-08-01 00:00:00 Labrador ~
##  9        893 MILEY       F             2008-07-01 00:00:00 Boxer
## 10        919 KENZI       F             2010-05-01 00:00:00 Schnauzer~
## # ... with 118,532 more rows, and 11 more variables: borough <chr>,
## #   zip_code <dbl>, community_district <dbl>, census_tract_2010 <dbl>,
## #   neighborhood_tabulation_area <chr>, city_council_district <dbl>,
## #   congressional_district <dbl>, state_senatorial_district <dbl>,
## #   license_issued_date <date>, license_expired_date <date>,
## #   borough_code <chr>
```

Which borough has the longest average licence duration?

7.8 HOW CAN I READ MY OWN TABULAR DATA INTO R?

7.8.1 WHAT IS TABULAR DATA?

Tabular data describes data that is in the form of a table: values arranged in rows each of the same length, or equivalently values arranged in columns each of the same length. Here's a small example of some tabular data:

site_id	latitude	longitude
DR-1	-49.85	-128.57
DR-3	-47.15	-126.72
MSK-4	-48.87	-123.40

Each row records information on a site at which water measurements are taken. There are three columns: a site identification code, and the location of the site in latitude and longitude.

This is an incredibly common way of *displaying* data, but when *storing* tabular data in a file, we need a way to communicate when records and values begin and end. A very popular format for doing this is CSV. CSV, is short for **c**omma **s**eparated **v**alues, and like the name suggests, a comma, `,`, is used to separate the values for each column, while each record goes on a new line. The file is plain text, but we use the extension `.csv` to indicate that is follows the CSV format conventions.

Here's how the table above would look inside the CSV file `site.csv`:

```
site_id,latitude,longitude
DR-1,-49.85,-128.57
DR-3,-47.15,-126.72
MSK-4,-48.87,-123.4
```

In this case the first line has the column names, this is common (and recommended!), but not universal.

Why is CSV so popular?

- It's human readable. CSV isn't a special file type, it is a simple plain text file that follows some conventions. This means you don't need any special software to look at the contents—you can open it up in anything that can examine text and take a look inside.
- It's computer readable. Because CSV files all have the same structure it's easy to write computer programs to read them. This means in almost any program designed to work with data, which is basically all the common programming languages, you'll find functions that will import CSV files. This also means it's easy to create CSV files—you can export them from Excel, write them from R, or even write one from scratch in a text editor.

If you want to look inside a CSV file in RStudio you can navigate to its location in the "Files" pane and click on its name. Selecting "View File", will open it in the Source pane.

However, if you want to work with CSV data in R, it isn't enough to look inside the file. You need to read the contents of the file and store it in R's memory. This process is known as data import.

7.8.2 IMPORTING CSV DATA INTO R

To work with data in R you need to have it in R's memory. The `read_csv()` function in the readr package will import a CSV file, and represent it as a tibble, if you give it the location of the CSV file. For example, to read the `site.csv` data and store it in an object called `sites`:

```
library(tidyverse)
sites <- read_csv(here("data", "site.csv"))
```

```
## Parsed with column specification:
## cols(
##   site_id = col_character(),
##   latitude = col_double(),
##   longitude = col_double()
## )
```

```
sites
```

```
## # A tibble: 3 x 3
##   site_id latitude longitude
##   <chr>      <dbl>     <dbl>
## 1 DR-1       -49.8     -129.
## 2 DR-3       -47.2     -127.
## 3 MSK-4      -48.9     -123.
```

FIXME: talk about files paths, point reader to the place where file paths are talked about, or assume file is in their working directory.

Notice that `read_csv()` gave us a message about what it did: it parsed our data file and found three columns `site_id`, `latitude` and

`longitude`. It also mentions what kind of data it assumed was in each column. FIXME: point to further discussion of data types.

The object `sites` is now R's representation of the data from the `site.csv` file.

7.8.3 EXERCISE: IMPORT `VISITED.CSV`

Use `read_csv()` to read `visited.csv` into R. How does R indicate a cell with a missing value?

```
visited <- read_csv(here("data", "visited.csv"))
```

```
## Parsed with column specification:
## cols(
##   visit_id = col_double(),
##   site_id = col_character(),
##   visit_date = col_date(format = "")
## )
```

```
visited
```

```
## # A tibble: 8 x 3
##   visit_id site_id visit_date
##      <dbl> <chr>   <date>
## 1      619 DR-1    1927-02-08
## 2      622 DR-1    1927-02-10
## 3      734 DR-3    1930-01-07
## 4      735 DR-3    1930-01-12
## 5      751 DR-3    1930-02-26
## 6      752 DR-3    NA
## 7      837 MSK-4   1932-01-14
## 8      844 DR-1    1932-03-22
```

7.8.4 EXERCISE: IMPORT IRS TAX RETURN DATA FOR NEW YORK CITY

Use `read_csv()` to import the CSV file `nyc-tax-returns.csv`.

FIXME: Should we introduce the "common things that go wrong" / "the most common additional arguments", e.g. `skip`, `na`, `col_names`, `col_types`? My feeling is not now, but sometime later.

7.9 HOW CAN I SAVE MY RESULTS?

Say, you've now got a summary of the license durations by borough:

```
dog_licenses %>%
  mutate(
    license_duration = license_expired_date - license_issued_date,
    borough_code = stringr::str_sub(neighborhood_tabulation_area, 1, 2)) %>%
  group_by(borough_code) %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

```
## # A tibble: 5 x 5
##   borough_code avg_duration  shortest_duration longest_duration n_licenses
##   <chr>        <drtn>        <drtn>            <drtn>                <int>
## 1 BK           465.2768 days 2 days            2191 days             29558
## 2 BX           435.8062 days 2 days            1919 days             12050
## 3 MN           494.0276 days 1 days            2189 days             41668
## 4 QN           452.1386 days 2 days            2186 days             24420
## 5 SI           439.4875 days 4 days            2164 days             10846
```

How do you save this result for future use?

If you just need this tibble later in your code you can assign it to a variable:

```
duration_by_borough <- dog_licenses %>%
  mutate(
    license_duration = license_expired_date - license_issued_date,
    borough_code = stringr::str_sub(neighborhood_tabulation_area, 1, 2)) %>%
  group_by(borough_code) %>%
  summarise(
    avg_duration = mean(license_duration),
    shortest_duration = min(license_duration),
    longest_duration = max(license_duration),
    n_licenses = n()
  )
```

Then you can access it whenever you need it:

```
duration_by_borough
```

```
## # A tibble: 5 x 5
##   borough_code avg_duration  shortest_duration longest_duration n_licenses
##   <chr>        <drtn>        <drtn>            <drtn>                <int>
## 1 BK           465.2768 days 2 days            2191 days             29558
## 2 BX           435.8062 days 2 days            1919 days             12050
## 3 MN           494.0276 days 1 days            2189 days             41668
## 4 QN           452.1386 days 2 days            2186 days             24420
## 5 SI           439.4875 days 4 days            2164 days             10846
```

This keeps our result around in memory, but often you also want to preserve the data in a file on disk. There are two common choices for format: CSV and RDS.

You've already seen CSV files. Saving your results in this format gives you all the benefits of that format: plain text files easily shared and opened. You can save a tibble as a CSV file with the readr function `write_csv()`, where all you need to specify is the path:

```
duration_by_borough %>% write_csv("duration-by-borough.csv")
```

You can then read this file in any project or R session with `read_csv()`:

```
duration_by_borough <- read_csv("duration-by-borough.csv")
```

```
## Parsed with column specification:
## cols(
##   borough_code = col_character(),
##   avg_duration = col_double(),
##   shortest_duration = col_double(),
##   longest_duration = col_double(),
##   n_licenses = col_double()
## )
```

RDS files are a special R format. They are binary files as opposed to plain text files, which means you can't just open them up and look

inside. If you are sharing them you'd also need your collaborators to have R. These are downsides, but the advantage of this format is that it can be much quicker to load, it will preserve special R data types (for example factors, or nested structures), and can save any R object not just tabular data structures.

To save the tibble as an RDS, use `write_rds()`:

```
duration_by_borough %>% write_rds("duration-by-borough.rds")
```

To read it back in, use `read_rds()`:

```
duration_by_borough <- read_rds("duration-by-borough.rds")
```

Often you'll save your data in both formats to make sure you get the best of both worlds. You'll talk more about where to save your data in FIXME: add link.

7.9.1 EXERCISE

Save the tibble with our extra columns:

```
dog_licenses %>%
  mutate(
    license_duration = license_expired_date - license_issued_date,
    borough_code = stringr::str_sub(neighborhood_tabulation_area, 1, 2))
```

into a CSV file called `dog-licenses-extra.csv`.

8 Publishing

8.1 QUESTIONS

- How can I share my work on the web?

8.2 WHY SHOULD I SHARE MY WORK ON THE INTERNET?

A key part of any project is communicating what you have learned. You've already seen how to create documents to communicate your work with R Markdown, and how to host whole projects on GitHub. This chapter is about sharing your work through a webpage on the internet. Some advantages of sharing your work on a webpage include:

- It's easy for your visitors—they just need to click on a link and see your work. They don't need to know anything about R Markdown, HTML, or GitHub.
- It provides a visually friendly and customizable landing point for people interested in your project. You can easily point them to the GitHub repo if they want more details.
- It's easy for you to make updates and those updates are immediately available to any visitors—you don't have to re-send anyone any files.

By the end of this chapter, you'll be able to take an R Markdown document that lives on your computer and share it with the world through a link to your own webpage.

8.3 WHAT DOES IT TAKE TO GET A WEBPAGE ONLINE?

To understand what it takes to get a webpage online, it helps to understand roughly what happens when you point your browser at a web address. When you point your browser at an address, for example https://merely-useful.github.io/r-publishing.html, the following things happen:

1. The URL `https://merely-useful.github.io` points to a location on another computer where the relevant files for this website reside. This computer is known as the website host or server, since it *hosts* the webpage files on its local storage drives and *serves* them to us via the internet.
2. The browser requests the file `r-publishing.html` from the host.
3. The browser reads the contents of `r-publishing.html` and displays it for you in the browser window.

In reverse order this process also describes what you need to do to get your own website online:

1. You need an HTML file that describes what people should see on your page.
2. You need to host the HTML file on a computer on the internet.
3. You need a way to associate a URL with the address of your host.

In this chapter, you'll learn a process for getting your work online that leverages what you already know—creating HTML files using R Markdown (#1 above) and how to host your work in a GitHub repository (#2). The third step will be handled by a GitHub service called GitHub Pages. By following the conventions that GitHub Pages expects, you'll be able to make a webpage for any of your repositories available at: `http://{{your_username}}.github.io/{{repo_name}}`.

8.4 HOW DO I GET MY WORK ON THE WEB?

8.4.1 A STARTING POINT

FIXME: revisit this later when more content is fleshed out. Maybe there will be a repo we can rely on all learners having that we can start from.

In practice you'll probably start thinking about a website once you've already done a lot of work on a project—your project will already have some analysis documented in R Markdown, be in version control, and hosted on GitHub. However, so that we can work with a specific example, you'll set up a project in this section that is less developed than where your project might be when you start thinking about making a website.

At the end of this section, you should have an RStudio project with an example report in `report.Rmd`. This project should also be on GitHub at `https://github.com/{{your_username}}/sharing-work`, where `{{your_username}}` should be substituted with your GitHub username, e.g. mine is at https://github.com/cwickham/sharing-work.

Let's start by creating a new project called, `sharing-work` by running the following in the RStudio console:

```
usethis::create_project("sharing-work")
```

Once the project opens, set it up to use version control by running the next code in the Console:

```
usethis::use_git()
```

Then add it as a repository on GitHub:

```
usethis::use_github()
```

Then so we have a report to work with, create a new R Markdown file ("File -> New File -> R Markdown"), making sure to leave the "Default Output Format" as HTML. Save this new file as `report.Rmd`.

Commit these changes and push your repository to GitHub. This is our starting point.

FIXME: Would it be better to get to this point by getting learners to fork a repo, then "New project -> From version control" in RStudio? Except forking isn't in the plan for the Version Control section.

8.4.2 HTML FILES

If you Knit `report.Rmd` you'll get `report.html`, an HTML document, because in the header of `report.Rmd` `output` is set to `html_document`. In the Files pane in RStudio if you click on `report.html`, you'll get two options: Open in Editor, or Display in Web Browser.

HTML is the language of webpages. If you "Open in Editor" you will see the contents of the file—it's plain text with markup tags to indicate how web browsers should display the text. If you "View in Web Browser" your browser will read, interpret and display the HTML for you. When you "View in Web Browser" you may notice the address bar in your browser looks something like:

```
file:///Users/wickhamc/Documents/Projects/sharing-work/docs/report.html
```

Just as `https://` is a signal that a file resides on a remote server computer, `file://` is a signal that a file lives on your computer locally. In spite of having "web" in their name, web browsers perfectly display local files as long as they are in a suitable format, such as HTML. However, you couldn't give this local address to someone else and expect it to work, because they don't have this file on their computer.

The HTML produced by R Markdown is completely self-contained, the browser needs no additional files to display the page as you see it now. So, you could email the file `report.html`, and your recipients could open it their browser and see the same result. However, our goal will be to put this HTML file on the web so you can share a link to the file instead of the file itself. You'll start by having this file accessible at the link `https://{{your_username}}.github.io/sharing-work/report.html`, then learn how to have it displayed with the shorter link `https://{{your_username}}.github.io/sharing-work`

8.4.3 SETTING UP YOUR REPO TO HAVE A WEB PAGE

Our goal is to get the report that is currently living in `report.html` displayed when a visitor heads to `https://{{your_username}}.github.io/sharing-work`

You'll set up GitHub Pages to look for your HTML files in the `docs` directory. Your first step will be to make this directory and put our report file inside. Create `docs/` with:

```
usethis::use_directory("docs")
```

Move `report.Rmd` and `report.html` into this directory. You can do this within the Files pane by checking the files and "More -> Move...". Or you can do it with code on the Console:

```
file.rename("report.Rmd", "docs/report.Rmd")
file.rename("report.html", "docs/report.html")
```

Commit and push your changes. Your repo should now have a structure like:

```
docs
    report.Rmd
    report.html
sharing-work.Rproj
```

Now you will activate GitHub Pages, so that GitHub will know to deliver your files when visitors head to: `https://{{your_username}}.github.io/sharing-work`

Visit your GitHub Repository and head to the "Settings" tab.

Scroll to the "GitHub Pages" section. Activate GitHub pages, with source set to "master branch /docs folder". You should see a message that your site is now live at: `https://{{your_username}}.github.io/sharing-work`

Try visiting: `https://{{your_username}}.github.io/sharing-work/report.html`. You should see your report. Congratulations you have a webpage!

8.4.4 GETTING A DEFAULT PAGE TO DISPLAY WHEN PEOPLE VISIT THE PROJECT SITE

You could send people the link, `https://{{your_username}}.github.io/sharing-work/report.html`, but it is often nicer to send them the shorter version without the file name: `https://{{your_username}}.github.io/sharing-work`. You can try this now, but it won't work—you'll see a message in your browser like: "404- File not found". This shorter URL points to a directory as opposed to a file. By default, when a server receives a request for a directory, it looks for a file to display with a default name—usually `index.html`. In your case there is no file called `index.html` so there is nothing to display.

If you would like the contents of `report.html` to be displayed as the homepage of your project, then rename `report.Rmd` to `index.Rmd`. You'll then need to regenerate the HTML file, commit it, and push your changes. For work that is communicated easily in one page, this would be a good option.

Alternatively, you might have a different page as the default page—one that summarizes the project and then links to other more detailed pages. You'll see how to do this over the next few sections. To get started create a new R Markdown document. Delete **all** the contents of the file, and then copy and paste in the following:

```
---
```

```
title: "Sharing work on a webpage"
author: "Me"
output: html_document
---

This project ...

* Read my report
* Visit this project on GitHub
```

Save the file as `index.Rmd`, knit it, commit, and push your changes. Your repo should now look like:

```
docs
    index.Rmd
    index.html
    report.Rmd
    report.html
sharing-work.Rproj
```

Now when you visit `https://{{your_username}}.github.io/sharing-work` you should see:

Note that it might take some time for the webpage to refresh automatically and detect the new index file. You can trigger this manually by going to GitHub Pages settings and changes the source branch to something else than `docs/` and then back again.

8.4.5 WHAT DOES IT TAKE TO GET YOUR WORK ON A WEBPAGE?

To sum up the process above, in its most minimal form, to have a webpage at `https://{{your_username}}.github.io/{{repo_name}}`, your repo at `https://github.com/{{your_username}}/{{repo_name}}` needs to:

1. have an `index.html` file in the `docs` directory (probably generated from `index.Rmd` in the same location), and
2. have GitHub Pages activated in repository settings with source set to "master branch /docs folder".

Be aware that everything inside the `docs` folder is now public, even if your repository is private.

You might have noticed this book lives at a GitHub Pages URL without a repo name—there is nothing after the `.io` in `https://merely-useful.github.io`. Both GitHub organizations and individuals can make use of this shorter address format (`merely-useful` is an organization rather than a user, so this is an organization site). There are a few differences between what you've learnt so far and the process of setting up a user site without a repo name (at `https://{{your_username}}.github.io`). First, you need to name your repository in a specific way—it must be called `{{your_username}}.github.io`. Second, user sites don't use the `docs/` folder—you put your HTML files at the top level in the repo. And third, you don't have to change any settings with user sites—GitHub will recognize the repo name and automatically serve it at `https://{{your_username}}.github.io`.

8.4.6 EXERCISE: CUSTOMIZE `INDEX.RMD`

- Edit `index.Rmd` to have your name as the author.
- Knit `index.Rmd` to verify your changes, then commit and push them.

- Visit `https://{{your_username}}.github.io/sharing-work` to check the updated site.

This is the workflow for making changes to your webpage. Make edits locally, and Knit to check them. Then commit and push to make those changes visible on the web.

8.5 HOW DO I LINK TO OTHER PAGES, FILES OR IMAGES?

8.5.1 LINKING TO OTHER PAGES

To create a link to another page in markdown file you use the syntax:

```
[text to display](url)
```

Once Knit to HTML, only `text to display` will be visible, and clicking on the text will take a viewer to `url`. For example, to add a link to my GitHub repo I might add the following line to `index.Rmd`:

```
Visit [my github repo](https://www.github.com/cwickham/sharing-work)
```

Which when Knit to HTML renders like:

> Visit my github repo

This is an example of an **absolute** URL. Just like when you specify file paths on your own computer, URLs can be both absolute and relative. An absolute URL describes a file location starting from and including the domain name. For instance, the absolute URL that points to `report.html` in my repo is `https://cwickham.github.io/sharing-work/report.html`.

Relative URLs are *relative* to the current HTML file. So, for instance if you are viewing `https://cwickham.github.io/sharing-work/index.html`, a relative link to my `report.html` would be `report.html` since this file is at the same level as `index.html` in my website structure. For pages created using this GitHub Pages workflow, your website structure is the same as the file structure in your `docs/` folder.

To add a link to `report.html` in `index.Rmd` I would add a line like:

```
See the [full report](report.html)
```

You should use relative URLs to reference any of **your** files (i.e. those in `docs/`). That way if you ever rename your repository, move it, or use a different hosting platform, your links will all work without changes. You must use absolute links for files that reside elsewhere on the internet.

8.5.2 EXERCISE: RELATIVE LINKS

Imagine your `docs/` folder had the following structure:

```
docs
    index.Rmd
    index.html
    diagrams
        workflow.png
|   reports
|       jan.Rmd
|       jan.html
|       feb.Rmd
|       feb.html
sharing-work.Rproj
```

Using a relative URL, how would you refer to:

- `jan.html` from `index.html`?
- `feb.html` from `jan.html`?
- `workflow.png` from `index.html`?

8.5.3 EXERCISE: ADD LINKS TO `INDEX.HTML`

Add to `index.Rmd`:

- a link to `report.html` using a relative link, and
- a link your GitHub repository using an absolute link.

8.5.4 LINKING TO SECTIONS WITHIN A PAGE

URLs can also refer to places inside the current page, most usually to another section. In R Markdown you've seen how to create headings using `#`. For example, an Appendix subsection might be:

```
## Appendix
```

If we want to link to this section from elsewhere, you prefix the section name with a single `#` in the URL. For example, if this section is in `report.Rmd` and you want to link to it elsewhere in `report.Rmd`, you could use:

```
See more details in the [appendix](#appendix)
```

The URL `#appendix` is interpreted as the heading with ID `appendix` in the current page. R Markdown creates IDs for all sections (and subsections) automatically, by converting to lower case and replacing spaces with dashes (`-`). But, you can explicitly set IDs too, by adding the ID with the `#` inside curly braces after the section heading. For instance you might prefer the shorter `appen` ID. You need to set it where the heading occurs:

```
## Appendix {#appen}
```

Then you can link to it using this shorter ID elsewhere:

```
See more details in the [appendix](#appen)
```

You can also use this strategy to link to sections in other pages by including the relative URL first. For instance, to refer to this "Appendix" section from `index.html` you could include in `index.Rmd`:

```
Some gory details of the analysis can also be found in the [Appendix of the report](report
```

8.5.5 EXERCISE: ADD AND LINK TO A SECTION IN `REPORT.RMD`

Add a new section to `report.Rmd` and include a link to it in `index.Rmd`. You'll need to Knit both `report.Rmd` and `index.Rmd`, and commit and push the HTML files to check your work.

8.5.6 INCLUDING IMAGES

Most of your images will likely be plots generated by R chunks in your R Markdown files and thus automatically included in the knitted HTML. If you want to display other images, you use the same syntax you saw in the Markdown section of the Reproducibility chapter chapter. That is, in your R Markdown file you'll include the image with something like:

```
![Image caption here.](path/to/image/file.png)
```

However, the `path/to/image/file.png` should be a relative URL pointing at an image in your `docs/` directory. For example, if you had an image, `me.png`, inside an `images` directory inside your `docs` folder:

```
docs
    index.Rmd
    index.html
    images
        me.png
    report.Rmd
    report.html
sharing-work.Rproj
```

You could include it in `index.html` by adding to `index.Rmd` the line:

```
![A picture of me](images/me.png)
```

Notice the syntax is very similar to adding a link to `me.png`:

```
[A picture of me](images/me.png)
```

Including the image displays the image inside at the appropriate place in the current page, *linking* to the image requires a viewer to click the link to see the image.

Your image will be included at full size, but you might find it too large. You can additionally specify some attributes for the image in curly braces immediately following the link. For instance, use the `width` attribute to set the image width, either in pixels:

```
[A picture of me](images/me.png){width=50} # default unit is px
```

Or as a percentage:

```
[A picture of me](images/me.png){width=50%}
```

8.5.7 EXERCISE: ADD AN IMAGE TO `INDEX.HTML`

Include an image in your `index.Rmd`. *(If you need an image to include you could always build your own version of an Octocat, GitHub's mascot).*

Don't forget, you'll need to commit both your re-Knit `index.html` and your image.

8.6 EXERCISE: ADD A WEBSITE TO AN EXISTING PROJECT

Add a website to one of your existing project repositories. You'll need to complete the following steps:

- Create a `docs/` directory in your project.
- Add an `index.Rmd` R Markdown document to the `docs/` folder and knit it to HTML to produce `index.html`.
- Commit these changes to git, and push to GitHub.
- Activate GitHub Pages in the repository settings with source set to "master branch /docs folder".
- Visit the site to check it is working.

A License

This is a human-readable summary of (and not a substitute for) the license. Please see https://creativecommons.org/licenses/by/4.0/legalcode for the full legal text.

This work is licensed under the Creative Commons Attribution 4.0 International license (CC-BY-4.0).

You are free to:

- **Share**—copy and redistribute the material in any medium or format
- **Remix**—remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution**—You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions**—You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

B Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

B.1 OUR STANDARDS

Examples of behavior that contributes to creating a positive environment include:

- using welcoming and inclusive language,
- being respectful of differing viewpoints and experiences,
- gracefully accepting constructive criticism,
- focusing on what is best for the community, and
- showing empathy towards other community members.

Examples of unacceptable behavior by participants include:

- the use of sexualized language or imagery and unwelcome sexual attention or advances,
- trolling, insulting/derogatory comments, and personal or political attacks,
- public or private harassment,
- publishing others' private information, such as a physical or electronic address, without explicit permission, and
- other conduct which could reasonably be considered inappropriate in a professional setting

B.2 OUR RESPONSIBILITIES

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

B.3 SCOPE

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

B.4 ENFORCEMENT

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing the project team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

B.5 ATTRIBUTION

This Code of Conduct is adapted from the Contributor Covenant version 1.4.

C Contributing

Contributions of all kinds are welcome. By offering a contribution, you agree to abide by our Code of Conduct and that your work may be made available under the terms of our license.

1. To report a bug or request a new feature, please check the list of open issues to see if it's already there, and if not, file as complete a description as you can.
2. If you have made a fix or improvement, please create a pull request. We will review these as quickly as we can (typically within 2-3 days). If you are tackling an issue that has already been opened, please name your branch `number-some-description` (e.g., `20-highlighting-active-block`) and put `Closes #N` (e.g., `Closes #20`) on a line by itself at the end of the PR's long description.

C.1 STYLE GUIDE

We follow the tidyverse style guide for R and PEP 8 for Python as closely as possible but specify some conventions further. We go against the style guides only when it is considered that it will improve clarity.

Specific conventions include:

- `variable_name` (snake_case)
- `function_name` and `method_name` (snake_case)
  - Please do *not* include empty parentheses to indicate a function, as this makes it hard to distinguish a function name from a call with no arguments.
- `folder-name/` (hyphens instead of underscores, trailing slash for clarity)
- `file-name` (hyphens instead of underscores)
- `'string'` and `"string"`
  - We will settle on single vs. double quotes before we publish :-)
- Method chaining in pandas:

```
(dataframe
    .method()
    .method(short_arg)
    .method(
        long_arg1,
        long_arg2))
```

For markdown, we use ATX-headers (`#` prefix) rather than Setext headers (`=/-` underlines), links with `[linkname][tag]` rather than `[linkname](url)`, and fenced code blocks rather than indented blocks.

There are more details for what we recommend for learners in rse-style.Rmd. Discuss further in issue #116.

Please note that we use Simplified English rather than Traditional English, i.e., American rather than British spelling and grammar.

C.2 SETTING UP

This book is written in Bookdown. If you want to preview builds on your own computer, please:

1. Follow the instructions for installing Bookdown.
2. Run `make everything` to recompile everything.
    - Run `make` on its own to see a list of targets for rebuilding specific volumes as HTML or PDF.

Please note that Bookdown works best with TinyTeX. After installing it, you can run `make tex-packages` to install all the packages this book depends on. You do *not* need to do this if you are only building and previewing the HTML versions of the books.

D Glossary

Abandonware FIXME
Absolute error FIXME
Absolute path FIXME
Accuracy FIXME

Action (in Make): FIXME

Active listening FIXME

Actual output (of a test): FIXME

Actual result FIXME
Aggregate FIXME
Agile development FIXME
Ally FIXME
Analysis and estimation FIXME

Annotated tag (in version control): FIXME

Append mode FIXME

Application Programming Interface (API): FIXME

Assertion FIXME
Authentic task A task which contains important elements of things that learners would do in real (non-classroom situations). To be authentic, a task should require learners to construct their own answers rather than choose between provided answers, and to work with the same tools and data they would use in real life.
Auto-completion FIXME
Automatic variable FIXME

Automatic variable (in Make): FIXME

Backlog FIXME
Bash FIXME
Beeswarm plot FIXME
Binary code FIXME
Bit rot FIXME
Boilerplate FIXME
Branch FIXME
Branch-per-feature workflow FIXME https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
Breakpoint FIXME
Buffer FIXME
Bug report FIXME
Bug tracker FIXME
Build tool FIXME
Build tool FIXME https://en.wikipedia.org/wiki/List_of_build_automation_software
Byte code FIXME
Call stack FIXME
Camel case FIXME

Catch (an exception): FIXME

Checking-driven development (CDD): FIXME

Checklist FIXME
Code browser FIXME
Code coverage FIXME
Cognitive load FIXME

Comma-separated values (CSV): FIXME

Command argument FIXME
Command flag FIXME
Command history FIXME
Command option FIXME
Command shell FIXME
Command switch FIXME

Command-line interface (CLI): FIXME

Comment FIXME

Commit hash FIXME
Commit message FIXME
Commit FIXME
Commons FIXME
Competent practitioner Someone who can do normal tasks with normal effort under normal circumstances. See also novice and expert.
Compiled language FIXME
Compiler FIXME
Computational competence FIXME
Computational notebook FIXME
Computational stylometry FIXME
Computational thinking FIXME
Conditional expression FIXME
Configuration object FIXME
Confirmation bias FIXME
Context manager FIXME
Continuation prompt FIXME
Continuous integration FIXME
Corpus FIXME
Coverage FIXME

Creative Commons - Attribution License (CC-BY): FIXME

Curb cuts https://medium.com/@mosaicofminds/the-curb-cut-effect-how-making-public-spaces-accessible-to-people-with-disabilities-helps-everyone-d69f24c58785
Current working directory FIXME
DRY (Don't Repeat Yourself) The general principle when programming that it's typically better to define something (a function, a constant…) once and refer to it consistently as a "single source of truth" throughout a piece of software than to define copies in multiple places, if only because then you only have to make any changes in one place. This is useful and helpful principle but should not be thought of as an involate rule.
Data engineering FIXME
Data package FIXME
Declarative programming FIXME
Default target FIXME

Default target (in Make): FIXME

Delimiter FIXME

Dependency graph FIXME
Design pattern FIXME
Design patterns FIXME
Destructuring FIXME
Dictionary FIXME

Digital Object Identifier (DOI): FIXME

Directory Tree If the nesting relationships between directories in a filesystem are drawn as arrows from the containing directory to the nested ones, a tree structure develops.
Directory A folder in a filesystem.
Disk Disk refers to disk storage, a physical component of a computer that stores information on a disk. The most common kind of disk storage is a hard disk drive, which is a storage drive with a non-removable disk.
Docstring FIXME
Documentation generator FIXME
Downvote FIXME
Embedded documentation FIXME
Eniac FIXME
Environment FIXME

Error (result from a unit test): FIXME

Escape sequence FIXME
Exception FIXME
Exit status FIXME

Expected output (of a test): FIXME

Expected result FIXME
Expert Someone who can diagnose and handle unusual situations, knows when the usual rules do not apply, and tends to recognize solutions rather than reasoning to them. See also competent practitioner and novice.
Exploratory programming FIXME
Exponent FIXME
External error FIXME

Failure (result from a unit test): FIXME

False beginner Someone who has studied a language before but is learning it again. False beginners start at the same point as true beginners (i.e., a pre-test will show the same proficiency) but can move much more quickly.
False negative FIXME
False positive FIXME
Feature boxing FIXME
Feature creep FIXME
Feature request FIXME
Filename extension FIXME
Filename stem FIXME
Filesystem Controls how files are stored and retrieved on disk by an operating system. Also used to refer to the disk that is used to store the files or the type of the filesystem.
Filter FIXME
Fixture FIXME
Flag variable FIXME
Flag FIXME
Folder FIXME
Forge FIXME
Fork FIXME
Format string FIXME

Frequently Asked Questions (FAQ): FIXME

Full identifier (in Git): FIXME

Fully-qualified name FIXME
Function attribute FIXME

Function (in Make): FIXME

GNU Public License (GPL): FIXME

Git branch FIXME
Git clone FIXME
Git conflict FIXME
Git fork FIXME
Git merge FIXME
Git pull FIXME

Git push FIXME
Git stage FIXME
Git FIXME
GitHub Pages FIXME
Globbing FIXME

Graphical user interface (GUI): FIXME

HTTP status code FIXME
Hitchhiker FIXME
Home directory FIXME
Hot spot FIXME
ISO date format FIXME
Impostor syndrome FIXME
In-place operator FIXME
Index FIXME
Install FIXME

Integrated Development Environment (IDE): FIXME

Internal error FIXME
Interpeter FIXME
Interpreted language FIXME
Interruption bingo FIXME
Issue tracking system FIXME
Issue FIXME

Iteration (in software development): FIXME

JSON FIXME

Jenny (a repository): FIXME

Join (of database tables): FIXME

Kebab case FIXME

Label (in issue tracker): FIXME

Learned helplessness FIXME
Library FIXME
Linter FIXME
List comprehension FIXME
Log file FIXME
Logging framework FIXME
Loop body FIXME

Loop (in Unix): FIXME

MIT License FIXME
Macro FIXME
Magic number FIXME
Magnitude FIXME
Makefile FIXME
Mantissa FIXME
Martha's Rules FIXME
Memory A physical device on your computer that temporarily stores information for immediate use.
Mental model A simplified representation of the key elements and relationships of some problem domain that is good enough to support problem solving.
Method A function that is specific to an object type, based on qualities of that type, e.g. a string method like `upper()` which turns characters in a string to uppercase.
Namespace A way of organizing names of related objects, functions, or variables to avoid confusion with (for instance) common names that might well occur in multiple packages.
Nano FIXME
Ngo FIXME

Not Invented Here (NIH): FIXME

Novice Someone who has not yet built a usable mental model of a domain. See also competent practitioner and expert.
ORCID FIXME
Object An object is a programming language's way of descrbing and storing values, usually labeled with a variable name.
Object-oriented programming FIXME
Open license FIXME
Open science FIXME

Operating system FIXME
Operational test FIXME
Oppression FIXME
Orthogonality FIXME
Overlay configuration FIXME
Overloading FIXME
Package FIXME
Pager FIXME
Pair programming FIXME
Parent directory FIXME
Parking lot FIXME
Path coverage FIXME
Path FIXME
Pattern rule FIXME
Pattern rule FIXME
Peer action FIXME
Phony target FIXME
Phony target FIXME

Pipe (in Unix): FIXME

Post-mortem FIXME
Pothole case FIXME
Precision FIXME

Prerequisite (in Make): FIXME

Privilege FIXME
Procedural programming FIXME
Process FIXME
Product manager FIXME
Project manager FIXME
Prompt FIXME
Provenance FIXME

Pseudorandom number generator (PRNG): FIXME

Public domain license (CC-0): FIXME

Pull request FIXME

Python FIXME
Raise FIXME

Raise (exception): FIXME

Raster image FIXME
Rebase FIXME
Recursion FIXME
Redirection FIXME
Refactor FIXME
Refactoring FIXME
Regular expression FIXME
Relative error FIXME
Relative import In Python, the importing of a module relative to the current path and thus likely from within the current package (e.g., `from . import generate`) rather than an import from a globally-defined package (e.g., `from zipfpy import generate`).
Relative path FIXME
Remote login FIXME
Remote repository FIXME
Repl FIXME
Repository FIXME

Representation State Transfer (REST): FIXME

Reproducible example (reprex): FIXME

Reproducible research FIXME

Research software engineer (RSE): FIXME

Restructured Text (reST) A plain text markup language used by much Python documentation and documentation tooling.
Revision FIXME
Root directory FIXME
Rotating file FIXME

Rule (in Make): FIXME

SSH key FIXME

SSH protocol FIXME

Scalable Vector Graphics (SVG): FIXME

Script FIXME

Seed (for pseudorandom number generator): FIXME

Semantic versioning FIXME https://semver.org/
Sense vote FIXME

Set and override (pattern): FIXME

Shebang FIXME
Shell script FIXME
Short circuit test FIXME

Short identifier (in Git): FIXME

Side effects FIXME
Sign FIXME
Silent error FIXME
Silent failure FIXME
Situational action FIXME
Snake case FIXME
Software development process FIXME
Source code FIXME
Stand-up meeting FIXME
Standard error FIXME
Standard error FIXME
Standard input FIXME
Standard input FIXME
Standard output FIXME
Standard output FIXME
Streaming data FIXME
Sturdy development FIXME
Subcommand FIXME
Subdirectory FIXME
Subsampling FIXME

Success (result from a unit test): FIXME

Sustainability FIXME
Sustainable software FIXME
Symbolic debugger FIXME
Syntax highlighting FIXME
Synthetic data FIXME
Tab completion FIXME

Tag (in version control): FIXME

Tag FIXME
Target FIXME

Target (in Make): FIXME

Target (of oppression): FIXME

Technical debt FIXME
Test coverage FIXME
Test framework FIXME
Test isolation FIXME
Test runner FIXME
Test-driven development FIXME
Three stickies FIXME
Ticket FIXME
Ticketing system FIXME
Tidy data As defined in Wickham (2014), tabular data is tidy if (1) each variable is in one column, (2) each different observation of that variable is in a different row, (3) there is one table for each kind of variable, and (4) if there are multiple tables, each includes a key so that related data can be linked.
Time boxing FIXME

Timestamp (on a file): FIXME

Tldr FIXME
Tolerance FIXME
Transitive dependency FIXME
Triage FIXME

Tuning FIXME
Tuple FIXME
Typesetting language FIXME
Unit test FIXME
Unix shell FIXME
Update operator See in-place operator.
Upvote FIXME
Validation FIXME

Variable (in Python): A symbolic name that reserves memory to store a value.

Variable FIXME

Variable (in Make): FIXME

Vector image FIXME
Verification FIXME
Violin plot FIXME
Virtual environment In Python, the `virtualenv` package allows you to create virtual, disposable, Python software environments containing only the packages and versions of packages you want to use for a particular project or task, and to install new packages into the environment without affecting other virtual environments or the system-wide default environment.

What You See Is What You Get (WYSIWYG): FIXME

Wildcard FIXME
Working directory FIXME
Working memory FIXME
Wrap code FIXME
Wrapper FIXME
YAML FIXME

E Novice R Learning Objectives

This appendix lays out the learning objectives for each set of lessons, and is intended to help instructors who want to use this curriculum.

FIXME: learning objectives for novice R.

F Novice R Key Points

FIXME: fill in keypoints for novice R

Conery, R. 2016. The imposter's handbook. Big Machine, Inc.
FIXME

Haddock, S., and C. Dunn. 2010. Practical computing for biologists. Sinauer Associates.
FIXME

Noble, W.S. 2009. A quick guide to organizing computational biology projects. *PLoS Computational Biology*. 5. doi:10.1371/journal.pcbi.1000424.
How to organize a small to medium-sized bioinformatics project.

Patil, P. et al. 2016. A statistical definition for reproducibility and replicability. doi:10.1101/066803.

Scopatz, A., and K.D. Huff. 2015. Effective computation in physics. O'Reilly Media.
A comprehensive introduction to scientific computing in Python

Taschuk, M., and G. Wilson. 2017. Ten simple rules for making research software more robust. *PLoS Computational Biology*. 13. doi:10.1371/journal.pcbi.1005412.
A short guide to making research software usable by other people.

Wickham, H. 2014. Tidy data. *Journal of Statistical Software*. 59. doi:10.18637/jss.v059.i10.
The defining paper on tidy data.

Wilson, G. et al. 2014. Best practices for scientific computing. *PLoS Biology*. 12. doi:10.1371/journal.pbio.1001745.
Outlines what a mature research software project should look like.

Wilson, G. et al. 2017. Good enough practices in scientific computing. *PLoS Computational Biology*. 13. doi:10.1371/journal.pcbi.1005510.
Outlines what a "good enough" research software project should look like.

Yenni, G.M. et al. 2019. Developing a modern data workflow for regularly updated data. *PLOS Biology*. 17:e3000125. doi:10.1371/journal.pbio.3000125.
FIXME