R for Data Science (V)

Koji Mizumura

August 20, 2018

Table of Contents

# Chapter 27: R Markdown

R Markdown provides an unified authoring framework for data science, combining your code, its results, and your prose commentary. R Markdown documents are fully reproducible and support dozens of output formats, like PDFs, Word files, slideshows, and more.

R Markdown files are designed to be used in three ways:

1. For communicating to decision makers, who want to focus on the conclusions, not the code behind the analysis.
2. For collaborating with other data scientists (including future you!), who are interested in both your conclusions, and how you reached them ( i.e. the code).
3. As an environment in which to do data science, as a modern day lab notebook where you can capture not only what you did, but also what you were thinking.

R Markdown integrates a number of R packages and external tools. This means that help is, by-and-large, not available through ?. Instead, as you work through this chapter, and use R Markdown in the future, keep these resources close to hand:

* R Markdown Cheat Sheet: Help > Cheatsheets > R Markdown Cheat Sheet,
* R Markdown Reference Guide: Help > Cheatsheets > R Markdown Reference Guide.

Both cheatsheets are also available at <http://rstudio.com/cheatsheets>.

## 27.2 R Markdown basics

This is an R Markdown file, a plain text file that has the extension .Rmd:

# ---  
# title: "Diamond sizes"  
# date: 2016-08-25  
# output: html\_document  
# ---  
#   
# ```{r setup, include = FALSE}  
# library(ggplot2)  
# library(dplyr)  
#   
# smaller <- diamonds %>%   
# filter(carat <= 2.5)  
# ```  
#   
# We have data about `r nrow(diamonds)` diamonds. Only   
# `r nrow(diamonds) - nrow(smaller)` are larger than  
# 2.5 carats. The distribution of the remainder is shown  
# below:  
#   
# ```{r, echo = FALSE}  
# smaller %>%   
# ggplot(aes(carat)) +   
# geom\_freqpoly(binwidth = 0.01)

It contains three important types of content:

1. An (optional) **YAML header** surrounded by ---s.
2. **Chunks** of R code surrounded by ```.
3. Text mixed with simple text formatting like # heading and \_italics\_.

When you open an .Rmd, you get a notebook interface where code and output are interleaved. You can run each code chunk by clicking the Run icon (it looks like a play button at the top of the chunk), or by pressing *Cmd/Ctrl+Shift+Enter*. RStudio executes the code and displays the results inline with the code:

To produce a complete report containing all text, code and results, click “Knit” or press *Cmd/Ctl+Shift+K* You can also do this programatically with rmarkdown::render("1-example.Rmd). This will display the report in the viewer pane, and create a self-contained HTML file that you can share with others.

When you **knit** the document, R Markdown sends the .Rmd file to knitr, <http://yihui.name/knitr/>, which executes all of the code chunks and creates a new markdown (.md) document which includes the code and its output. The markdown file generated by knitr is then processed by pandoc, <http://pandoc.org/>, which is responsible for creating the finished file. The advantage of this two step workflow is that you can create a very wide range of output formats, as you’ll learn about in R markdown formats.

To get started with your own .Rmd file, select File > New File > R Markdown… in the menubar. RStudio will launch a wizard that you can use to pre-populate your file with useful content that reminds you how the key features of R Markdown work.

The following sections dive into the three components of an R Markdown document in more details: the markdown text, the code chunks, and the YAML header.

## 27.3 Text fomating with Markdown

Prose in .Rmd files is written in Markdown, a lightweight set of conventions for formatting plain text files. Markdown is designed to be easy to read and easy to write. It is also very easy to learn. The guide below shows how to use Pandoc’s Markdown, a slightly extended version of Markdown that R Markdown understands.

# Text formatting   
# ------------------------------------------------------------  
#   
# \*italic\* or \_italic\_  
# \*\*bold\*\* \_\_bold\_\_  
# `code`  
# superscript^2^ and subscript~2~  
#   
# Headings  
# ------------------------------------------------------------  
#   
# # 1st Level Header  
#   
# ## 2nd Level Header  
#   
# ### 3rd Level Header  
#   
# Lists  
# ------------------------------------------------------------  
#   
# \* Bulleted list item 1  
#   
# \* Item 2  
#   
# \* Item 2a  
#   
# \* Item 2b  
#   
# 1. Numbered list item 1  
#   
# 1. Item 2. The numbers are incremented automatically in the output.  
#   
# Links and images  
# ------------------------------------------------------------  
#   
# <http://example.com>  
#   
# [linked phrase](http://example.com)  
#   
# ![optional caption text](path/to/img.png)  
#   
# Tables   
# ------------------------------------------------------------  
#   
# First Header | Second Header  
# ------------- | -------------  
# Content Cell | Content Cell  
# Content Cell | Content Cell

The best wat to learn is simply try them out. It will take a few days, but soon they will become second nature, and you won’t need hink about them. If you forget, you can get to a handy reference sheet with *Help > Markdown Quick Reference*.

## 27.4 Code chunks

To run code inside an R Markdown document, you need to insert a chunk. There are three ways to do so:

1. The keyboard shortcut *Cmd/Ctrl + Alt + I*
2. The “Insert” button icon in the editor toolbar.
3. By manually typing the chunk delimiters {r} and.

Obviously, I’d recommend you learn the keyboard shortcut. It will save you a lot of time in the long run!

You can continue to run the code using the keyboard shortcut that by now (I hope!) you know and love: *Cmd/Ctrl + Enter*. However, chunks get a new keyboard shortcut: *Cmd/Ctrl + Shift + Enter*, which runs all the code in the chunk. Think of a chunk like a function. A chunk should be relatively self-contained, and focussed around a single task.

The following sections describe the chunk header which consists of ```{r, followed by an optional chunk name, followed by comma separated options, followed by }. Next comes your R code and the chunk end is indicated by a final.

### 27.4.1 Chunk name

Chunks can be give an optional name {r by-name}. This has three advantages: 1. You can more easily navigate to specific chunks using the drop-down code navigator in the bottom-left of the script editor: 2. Graphics produced by the chunks will have useful names that make them easier to use elsewhere. More on that in other important options. 3. You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that below.

There is one chunk name that imbues special behaviour: setup. When you’re in a notebook mode, the chunk named setup will be run automatically once, before any other code is run.

### 27.4.2 Chunk options

Chunk output can be customised with options, arguments supplied to chunk header. Knitr provides almost 60 options that you can use to customize your code chunks. Here we’ll cover the most important chunk options that you’ll use frequently. You can see the full list at <http://yihui.name/knitr/options/>.

The most important set of options controls if your code block is executed and what results are inserted in the finished report:

* eval = FALSE prevents code from being evaluated. (And obviously if the code is not run, no results will be generated). This is useful for displaying example code, or for disabling a large block of code without commenting each line.
* include = FALSE runs the code, but doesn’t show the code or results in the final document. Use this for setup code that you don’t want cluttering your report.
* echo = FALSE prevents code, but not the results from appearing in the finished file. Use this when writing reports aimed at people who don’t want to see the underlying R code.
* message = FALSE or warning = FALSE prevents messages or warnings from appearing in the finished file.
* results = 'hide' hides printed output; fig.show = ‘hide’ hides plots.
* error = TRUE causes the render to continue even if code returns an error. This is rarely something you’ll want to include in the final version of your report, but can be very useful if you need to debug exactly what is going on inside your .Rmd. It’s also useful if you’re teaching R and want to deliberately include an error. The default, error = FALSE causes knitting to fail if there is a single error in the document.

### 27.4.3 Table

By default, R Markdown prints data frames and matrices as you’d see them in the console:

mtcars[1:5,]

## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

If you prefer that data be displayed with additional formattin you can use the knitr::kable function. The code below generates the Table 27.1

knitr::kable(  
 mtcars[1:5,],  
 caption="A knitr kable"  
)

A knitr kable

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
| Mazda RX4 | 21.0 | 6 | 160 | 110 | 3.90 | 2.620 | 16.46 | 0 | 1 | 4 | 4 |
| Mazda RX4 Wag | 21.0 | 6 | 160 | 110 | 3.90 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| Datsun 710 | 22.8 | 4 | 108 | 93 | 3.85 | 2.320 | 18.61 | 1 | 1 | 4 | 1 |
| Hornet 4 Drive | 21.4 | 6 | 258 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| Hornet Sportabout | 18.7 | 8 | 360 | 175 | 3.15 | 3.440 | 17.02 | 0 | 0 | 3 | 2 |

Read the documentation for ?knitr::kable to see the other ways in which you can customise the table. For even deeper customisation, consider the xtable, stargazer, pander, tables, and ascii packages. Each provides a set of tools for returning formatted tables from R code.

There is also a rich set of options for controlling how figures are embedded. You’ll learn about these in [saving your plots](http://r4ds.had.co.nz/graphics-for-communication.html#saving-your-plots)

### 27.4.4 Caching

Normally, each knit of a document starts from a completely clean slate. This is great for reproducibility, because it ensures that you’ve captured every important computation in code. However, it can be painful if you have some computations that take a long time. The solution is cache = TRUE. When set, this will save the output of the chunk to a specially named file on disk. On subsequent runs, knitr will check to see if the code has changed, and if it hasn’t, it will reuse the cached results. .

rawdata <- readr::read\_csv("a\_very\_large\_file\_csv")

processed\_data <- rawdata %>%   
 filter(!is.na(import\_var)) %>%   
 mutate(new\_variable = complicated\_transformation(x, y, z))

Caching the processed\_data chunk means that it will get re-run if the dplyr pipeline is changed, but it won’t get rerun if the `read\_csv() call changes. You can avoid the probelem with the dependson chunk option:

processed\_data <- rawdata %>%   
 filter(!is.na(import\_var)) %>%   
 mutate(new\_variable=complicated\_transformation(x,y,z))

dependson should contain a character vector of *every* chunk that the cached chunk depends on. Knitr will update the results for the cached chunk whenever it detects that one of its dependencies have changed.

Note that the chunks won’t update if a\_very\_large\_file.csv changes, because knitr caching only tracks changes within the .Rmd file. If you want to also track changes to that file, you can use the cache.extrac option.

This is an arbitrary R expression that will invalidate the cache whenever it changes. A good function to use is file.info(): it returns a bunch of information about the file including when it was last modified. Then you can write:

rawdata <- readr::read\_csv("a\_very\_large\_file.csv")

As your caching strategies get progressively more complicated, it’s a good idea to regularly clear out all your caches with knitr::clean\_cache().

I’ve used the advice of David Robinson to name these chunks: each chunk is named after the primary object that it creates. This makes it easier to understand the dependson specification.

### 27.4.5 Global options

As you work more with knitr, you will discover that some of the default chunk options don’t fit your needs and you want to change them. You can do this by calling knitr::opts\_chunk$set() in a code chunk. For example, when writing books and tutorials I set:

knitr::opts\_chunk$set(  
 comment = "#>",  
 collapse = TRUE  
)

This uses my preferred comment formatting, and ensures that the code and output are kept closely entwined. On the other hand, if you were preparing a report, you might set:

knitr::opts\_chunk$set(  
 echo = FALSE  
)

That will hide the code by default, so only showing the chunks you deliberately choose to show (with echo = TRUE). You might consider setting message = FALSE and warning = FALSE, but that would make it harder to debug problems because you wouldn’t see any messages in the final document.

### 27.4.6 Inline code

There is one other way to embed R code into an R Markdown document: directly into the text, with: r. This can be very useful if you mention properties of your data in the text. For example, in the example document I used at the start of the chapter I had:

library(tidyverse)  
#> Warning: package 'tidyverse' was built under R version 3.5.1  
#> -- Attaching packages ------------------------- tidyverse 1.2.1 --  
#> √ ggplot2 3.0.0 √ purrr 0.2.5  
#> √ tibble 1.4.2 √ dplyr 0.7.6  
#> √ tidyr 0.8.1 √ stringr 1.3.1  
#> √ readr 1.1.1 √ forcats 0.3.0  
#> Warning: package 'ggplot2' was built under R version 3.5.1  
#> Warning: package 'purrr' was built under R version 3.5.1  
#> Warning: package 'dplyr' was built under R version 3.5.1  
#> -- Conflicts ---------------------------- tidyverse\_conflicts() --  
#> x dplyr::filter() masks stats::filter()  
#> x dplyr::lag() masks stats::lag()

We have data about 53940 diamonds. Only 2.69710^{4} are larger than 2.5 carats. The distribution of the remainder is shown below:

When the report is knit, the results of these computations are inserted into the text: > We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

When inserting numbers into text, format() is your friend. It allows you to set the number of digits so you don’t print to a ridiculous degree of accuracy, and a big.mark to make numbers easier to read.

I’ll often combine these into a helper function:

comma <- function(x) format(x,digits=2,big.mark=",")  
comma(3452345)  
#> [1] "3,452,345"  
comma(.12358124331)  
#> [1] "0.12"

### 27.4.7 Exercises

## 27.5 Trouble shooting

Troubleshooting R Markdown documents can be challenging because you are no longer in an interactive R environment, and you will need to learn some new tricks. The first thing you should always try is to recreate the problem in an interactive session. Restart R, then “Run all chunks” (either from Code menu, under Run region), or with the keyboard shortcut *Ctrl + Alt + R*. If you’re lucky, that will recreate the problem, and you can figure out what’s going on interactively.

If that doesn’t help, there must be something different between your interactive environment and the R markdown environment. You’re going to need to systematically explore the options. The most common difference is the working directory: the working directory of an R Markdown is the directory in which it lives. Check the working directory is what you expect by including getwd() in a chunk.

getwd()  
#> [1] "C:/Users/kojikm.mizumura/Desktop/Data Science/R for Data Science"

Next, brainstorm all the things that might cause the bug. You’ll need to systematically check that they’re the same in your R session and your R markdown session. The easiest way to do that is to set error = TRUE on the chunk causing the problem, then use print() and str() to check that settings are as you expect.

## 27.6 YAML header

You can control many other “whole document” settings by tweaking the parameters of the YAML header. You might wonder what YAML stands for: it’s “yet another markup language”, which is designed for representing hierachical data in a way that’s easy for humans to read and write. R Markdown uses it control many details of the output. Here we’ll discuss two: document parameters and bibligraphies

### 27.6.1 Parameters

R Markdown documents can include one or more parameters whose values can be set when you render the report. Parameters are useful when you want to re-render the same report per branch, exam results by student, or demographic summaries by country. To declare one or more parameters, use the params field.

This example uses a my\_class parameter to determine which colass of cars to display:

# ---  
# output: html\_document  
# params:  
# my\_class: "suv"  
# ---  
# ```{r setup, include = FALSE}  
# library(ggplot2)  
# library(dplyr)  
#   
# class <- mpg %>% filter(class == params$my\_class)  
# ```  
#   
# # Fuel economy for `r params$my\_class`s  
#   
# ```{r, message = FALSE}  
# ggplot(class, aes(displ, hwy)) +   
# geom\_point() +   
# geom\_smooth(se = FALSE)  
# ```

As you can see, parameters are available within the code chunks as a read-only list named params. You can write atomic vectors directly into the YAML header. You can also run arbitrary R expressions by prefacing the parameter value with !r. This is a good way to specify date/time parameters.

params:  
 start:!r lubridate::ymd("2015-01-01")  
 snapshot: !r lubridate::ymd\_hms("2015-01-01 12:30:00")

In RStudio, you can click the “Knit with Parameters” option in the Knit dropdown menu to set parameters, render, and preview the report in a single user friendly step. You can customise the dialog by setting other options in the header. See more details <http://rmarkdown.rstudio.com/developer_parameterized_reports.html#parameter_user_interfaces>

Alternatively, if you need to produce many such parameterised reports, you can call rmarkdown::render() with a list of params:

rmarkdown::render("fuel-economy.Rmd", params = list(my\_class = "suv"))

This is particularly powerful in conjunction with purrr:pwalk(). The following example creates a report for each value of class found in mpg. First we create a data frame that has one row for each class, giving the filename of the report and the params:

reports <- tibble(  
 class=unique(mpg$class),  
 filename=stringr::str\_c("fuel-economy-",class,"html"),  
 params=purrr::map(class,~list(my\_class=.))  
)  
reports  
#> # A tibble: 7 x 3  
#> class filename params   
#> <chr> <chr> <list>   
#> 1 compact fuel-economy-compacthtml <list [1]>  
#> 2 midsize fuel-economy-midsizehtml <list [1]>  
#> 3 suv fuel-economy-suvhtml <list [1]>  
#> 4 2seater fuel-economy-2seaterhtml <list [1]>  
#> 5 minivan fuel-economy-minivanhtml <list [1]>  
#> 6 pickup fuel-economy-pickuphtml <list [1]>  
#> 7 subcompact fuel-economy-subcompacthtml <list [1]>

Then we match the column names to the argument names of render(), and use purrr’s parallel walk to call render() once for each row:

reports %>%   
 select(output\_file=filename,params) %>%   
 purrr::pwalk(rmarkdown::render,input="fuel-economy.Rmd")

### 27.6.2 Bibliographies and Citations

Pandoc can automatically generate citations and a bibliography in a number of styles. To use this feature, specify a bibliography file using the bibliography field in your file’s header. The field should contain a path from the directory that contains your .Rmd file to the file that contains the bibliography file:

bibliography: rmarkdown.bib

You can use many common bibliography formats including BibLaTeX, BibTeX, endnote, medline. To create a citation within your.Rmd file, use a key composed of ‘@’ + citation identifier from the bibligraphy file. Then place the citation in square brackets. Here are some examples:

bibliography: rmarkdown.bib  
csl: apa.csl

As with the bibliography field, your csl file should contain a path to the file. Here I assume that the csl file is in the same directory as the .Rmd file. A good place to find CSL style files for common bibliography styles is <http://github.com/citation-style-language/styles>.

## 27.7 Learning more

R Markdown is still relatively young, and is still growing rapidly. The best place to stay on top of innovations is the official R Markdown website: <http://rmarkdown.rstudio.com>.

There are two important topics that we haven’t covered here: collaboration, and the details of accurately communicating your ideas to other humans. Collaboration is a vital part of modern data science, and you can make your life much easier by using version control tools, like Git and GitHub. We recommend two free resources that will teach you about Git:

1. “Happy Git with R”: a user friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online: <http://happygitwithr.com>
2. The **“Git and GitHub”** chapter of R Packages, by Hadley. You can also read it for free online: <http://r-pkgs.had.co.nz/git.html>.

I have also not touched on what you should actually write in order to clearly communicate the results of your analysis. To improve your writing, I highly recommend reading either Style: Lessons in Clarity and Grace by Joseph M. Williams & Joseph Bizup, or The Sense of Structure: Writing from the Reader’s Perspective by George Gopen. Both books will help you understand the structure of sentences and paragraphs, and give you the tools to make your writing more clear. (These books are rather expensive if purchased new, but they’re used by many English classes so there are plenty of cheap second-hand copies). George Gopen also has a number of short articles on writing at <https://www.georgegopen.com/the-litigation-articles.html>. They are aimed at lawyers, but almost everything applies to data scientists too.

# Chapter 28: Graphics for communication

## 28.1 Introduction

In [exploratory data analysis](http://r4ds.had.co.nz/exploratory-data-analysis.html#exploratory-data-analysis), you learned how to use plots as tools for exploration. When you make exploratory plots, you know—even before looking—which variables the plot will display. You made each plot for a purpose, could quickly look at it, and then move on to the next plot. In the course of most analyses, you’ll produce tens or hundreds of plots, most of which are immediately thrown away.

Now that you understand your data, you need to communicate your understanding to others. Your audience will likely not share your background knowledge and will not be deeply invested in the data. To help others quickly build up a good mental model of the data, you will need to invest considerable effort in making your plots as self-explanatory as possible. In this chapter, you’ll learn some of the tools that ggplot2 provides to do so.

This chapter focuses on the tools you need to create good graphics. I assume that you know what you want, and just need to know how to do it. For that reason, I highly recommend pairing this chapter with a good general visualisation book. I particularly like [The Truthful Art](https://www.amazon.com/dp/0321934075/ref=cm_sw_su_dp), by Albert Cairo. It doesn’t teach the mechanics of creating visualisations, but instead focuses on what you need to think about in order to create effective graphics.

### 28.1.1 Prerequisites

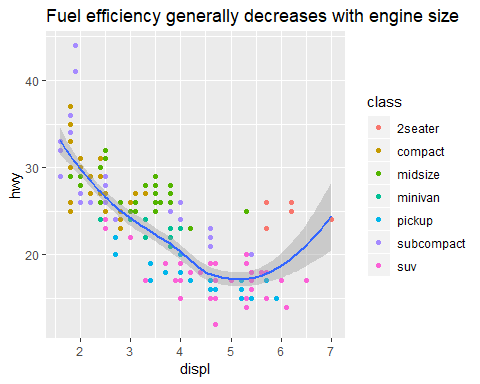
In this chapter, we’ll focus once again on ggplot2. We’ll also use a little dplyr for data manipulation, and a few ggplot2 extension packages, including ggrepel and viridis. Rather than loading those extensions here, we’ll refer to their functions explicitly, using the :: notation. This will help make it clear which functions are built into ggplot2, and which come from other packages. Don’t forget you’ll need to install those packages with install.packages() if you don’t already have them.

library(tidyverse)

## 28.2 Label

The easiest place to start when turning an exploratory graphic into an expository graphic is with good labels. You add labels with labs() function. This example adds a plot titile:

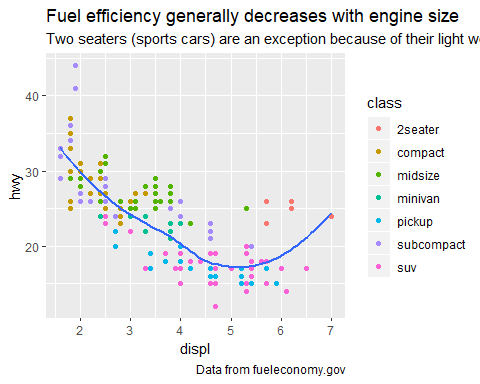
# ?mpg  
ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth(se=T)+  
 labs(title="Fuel efficiency generally decreases with engine size")  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



The purpose of the plot title is to summarise the main finding. Avoid titles that just describe what the plot is, e.g., “A scatterplot of engine displacement vs fuel economy”

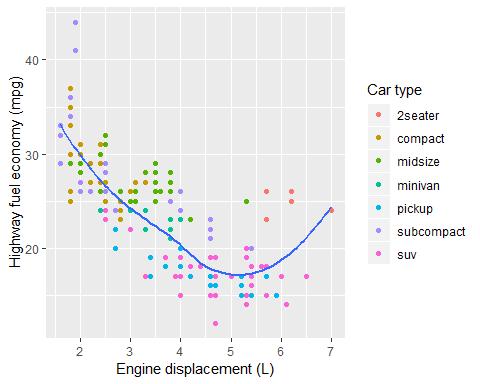
If you need to add more text, there are two other useful labels you can use in ggplot2 2.2.0 and above (what should be available by the time you’re reading this book): - subtitle addes additional detail in smaller font beneath the title - caption adds text at the bottom right of the plot, often used describe the source of the data.

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth(se=F)+  
 labs(  
 title = "Fuel efficiency generally decreases with engine size",  
 subtitle = "Two seaters (sports cars) are an exception because of their light weight",  
 caption = "Data from fueleconomy.gov"  
 )  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



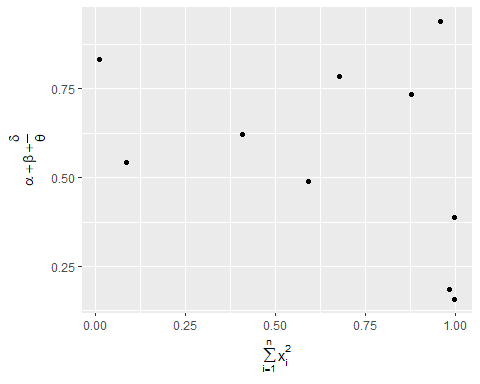
You can also use labs() to replace the axis and legend titles. It’s usually a good idea to replace short variable names with more detailed descriptions, and to include the units.

ggplot(mpg, aes(displ, hwy)) +  
 geom\_point(aes(colour = class)) +  
 geom\_smooth(se = FALSE) +  
 labs(  
 x = "Engine displacement (L)",  
 y = "Highway fuel economy (mpg)",  
 colour = "Car type"  
 )  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'

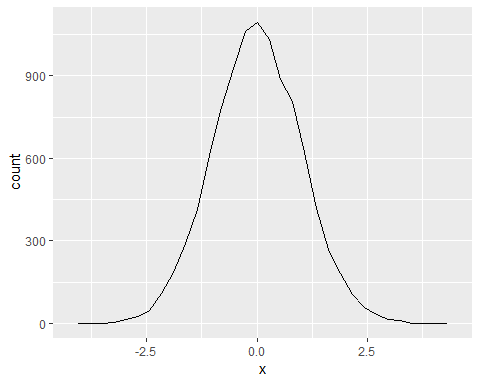


It’s possible to use mathematical equations instead of text strings. Just switch “” out for quote() and read about the available options in?plotmath:

df <- tibble(  
 x=runif(10),  
 y=runif(10)  
)  
head(df,3)  
#> # A tibble: 3 x 2  
#> x y  
#> <dbl> <dbl>  
#> 1 0.984 0.186  
#> 2 0.593 0.489  
#> 3 0.999 0.387  
  
ggplot(df,aes(x,y))+  
 geom\_point()+  
 labs(  
 x=quote(sum(x[i]^2,i==1,n)),  
 y=quote(alpha+beta+frac(delta,theta))  
 )



# normal distribution  
df1 <- tibble(  
 x=rnorm(10000))  
head(df1)  
#> # A tibble: 6 x 1  
#> x  
#> <dbl>  
#> 1 -0.666   
#> 2 0.237   
#> 3 -0.387   
#> 4 0.771   
#> 5 -0.0121  
#> 6 -1.04  
ggplot(data=df1,aes(x=x))+  
 geom\_freqpoly()  
#> `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.

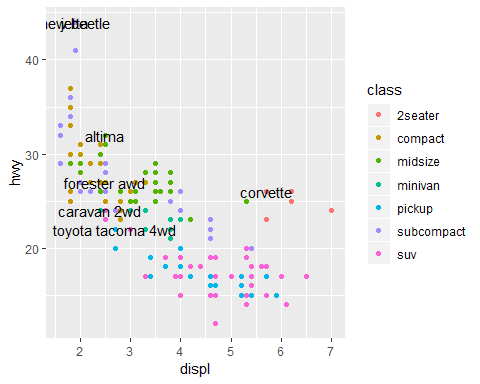


## 28.3 Annotations

In addition to labelling major components of your plot, it’s often useful to label individual observations or groups of observations. The first tool you have at your disposal is geom\_text(). geom\_text() is similar to geom\_point(), but it has an additional aesthetic: label. This makes it possible to add textual labels to your plots.

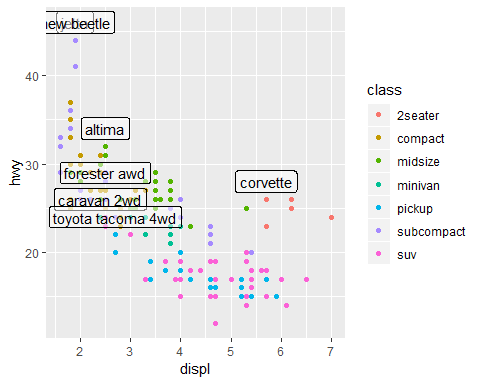
There are two possible sources of labels. First, you might have a tibble that provides labels. The plot below isn’t terribly useful, but it illustrates a useful approach: pull out the most efficient car in each class with dplyr, and then label it on the plot:

best\_in\_class <- mpg %>%   
 group\_by(class) %>%   
 filter(row\_number(desc(hwy))==1)  
head(best\_in\_class)  
#> # A tibble: 6 x 11  
#> # Groups: class [6]  
#> manufacturer model displ year cyl trans drv cty hwy fl class  
#> <chr> <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>  
#> 1 chevrolet corv~ 5.7 1999 8 manu~ r 16 26 p 2sea~  
#> 2 dodge cara~ 2.4 1999 4 auto~ f 18 24 r mini~  
#> 3 nissan alti~ 2.5 2008 4 manu~ f 23 32 r mids~  
#> 4 subaru fore~ 2.5 2008 4 manu~ 4 20 27 r suv   
#> 5 toyota toyo~ 2.7 2008 4 manu~ 4 17 22 r pick~  
#> 6 volkswagen jetta 1.9 1999 4 manu~ f 33 44 d comp~  
  
ggplot(mpg,aes(displ, hwy))+  
 geom\_point(aes(colour=class))+  
 geom\_text(aes(label=model),data=best\_in\_class)



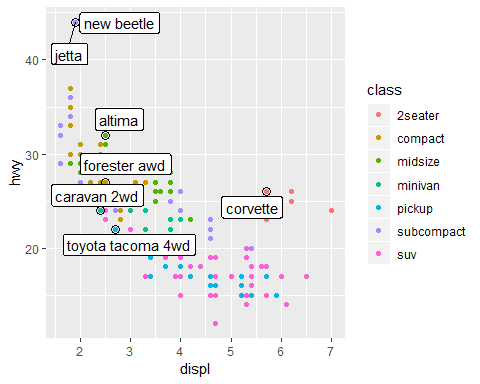
This is hard to read because the labels overlap with each other, and with the points. We can make things a little better by switching to geom\_label() which draws a rectangle behind the text. We also use the nudge\_y parameter to move the labels slightly above the corresponding points:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))+  
 geom\_label(aes(label=model),data=best\_in\_class,nudge\_y = 2,alpha=0.5)



That helps a bit, but if you look closely in the top-left hand corner, you’ll notice that there are two labels practically on top of each other. This happens because the highway mileage and displacement for the best cars in the compact and subcompact categories are exactly the same. There’s no way that we can fix these by applying the same transformation for every label. Instead, we can use the **ggrepel** package by Kamil Slowikowski. This useful package will automatically adjust labels so that they don’t overlap:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))+  
 geom\_point(size=3,shape=1,data=best\_in\_class)+  
 ggrepel::geom\_label\_repel(aes(label=model),data=best\_in\_class)

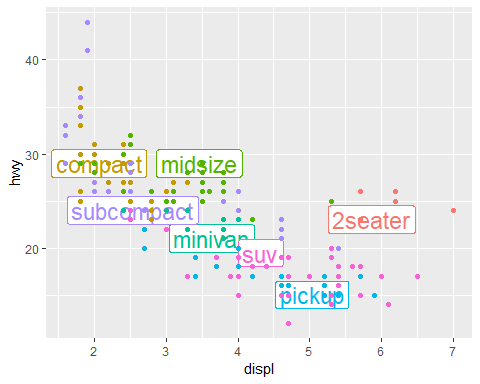


Note another handy technique used here: I added a second layer of large, hollow points to highlight the points that I’ve labelled.

You can sometimes use the same idea to replace the legend with labels placed directly on the plot. It’s not wonderful for this plot, but it isn’t too bad. (theme(legend.position = "none") turns the legend off — we’ll talk about it more shortly.)

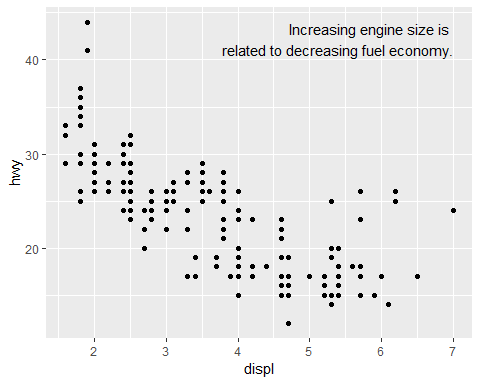
class\_avg <- mpg %>%   
 group\_by(class) %>%   
 summarise(  
 displ=median(displ),  
 hwy=median(hwy)  
 )  
  
class\_avg  
#> # A tibble: 7 x 3  
#> class displ hwy  
#> <chr> <dbl> <dbl>  
#> 1 2seater 6.2 25   
#> 2 compact 2.2 27   
#> 3 midsize 2.8 27   
#> 4 minivan 3.3 23   
#> 5 pickup 4.7 17   
#> 6 subcompact 2.2 26   
#> 7 suv 4.65 17.5

ggplot(mpg,aes(displ,hwy,colour=class))+  
 ggrepel::geom\_label\_repel(aes(label=class),  
 data=class\_avg,  
 size=6,  
 label.size=0,  
 segment.color=NA)+  
 geom\_point()+  
 theme(legend.position = "none")



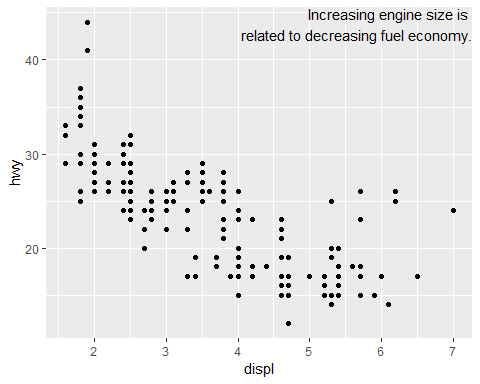
Alternatively, you might just want to add a single label to the plot, but you’ll still need to create a data frame. Often, you want the label in the corner of the plot, so it’s convenient to create a new data frame using summarise() to compute the maximum values of x and y.

label <- mpg %>%   
 summarise(  
 displ=max(displ),  
 hwy=max(hwy),  
 label="Increasing engine size is \n related to decreasing fuel economy.")  
   
ggplot(mpg,aes(displ,hwy))+  
 geom\_point()+  
 geom\_text(aes(label=label),data=label,vjust="top",hjust="right")



I yo uwant to place the text exactly on the borders of the plot, you can use +Inf and -Inf., Since we’re no longer computing the position from mpg, we can use tibble() to create the data frame:

label <- tibble(  
 displ = Inf,  
 hwy = Inf,  
 label = "Increasing engine size is \nrelated to decreasing fuel economy."  
)  
  
ggplot(mpg, aes(displ, hwy)) +  
 geom\_point() +  
 geom\_text(aes(label = label), data = label, vjust = "top", hjust = "right")



"Increasing engine size is related to decreasing fuel economy." %>%  
 stringr::str\_wrap(width = 40) %>%  
 writeLines()  
#> Increasing engine size is related to  
#> decreasing fuel economy.  
#> Increasing engine size is related to  
#> decreasing fuel economy

Note the use of hjust and vjust to control the alignment of the label. Figure 28.1 shows all nine possible combinations.

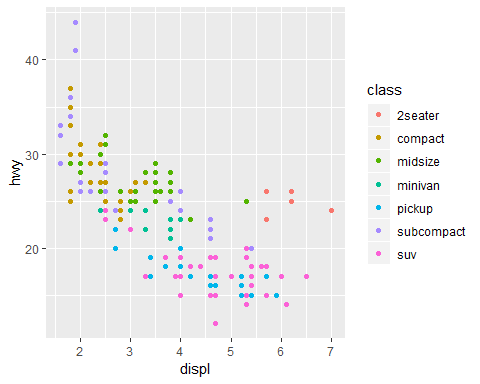
Remember, in addition to geom\_text(), you have many other geoms in ggplot2 available to help annotate your plot. A few ideas: - Use geom\_hline() and geom\_vline() to add reference lines. I often make them thick (size = 2) and white (colour = white), and draw them underneath the primary data layer. That makes them easy to see, without drawing attention away from the data.

* Use geom\_rect() to draw a rectangle around points of interest. The boundaries of the rectangle are defined by aesthetics xmin, xmax, ymin, ymax.
* Use geom\_segment() with the arrow argument to draw attention to a point with an arrow. Use aesthetics x and y to define the starting location, and xend and yend to define the end location.

## 28.4 Scales

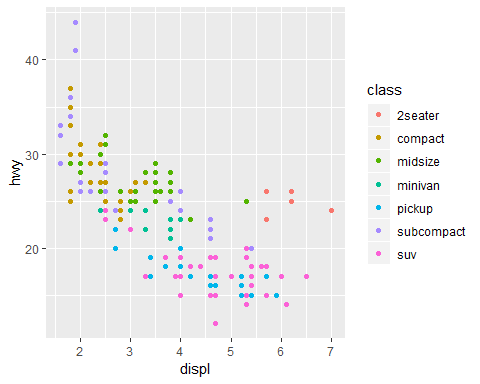
The third way you can make your plot better for communication is to adjust the scales. Scales control the mapping from data values to things that you can perceive. Normally, ggplot2 automatically adds scales for your. For example, when you type:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))



ggplo2 automatically adds default scales behind the scenes:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))+  
 scale\_x\_continuous()+  
 scale\_y\_continuous()+  
 scale\_colour\_discrete()



Note the naming scheme for scales: scale\_ followed by the name of the aesthetic, then \_, then the name of the scale. The default scales are named according to the type of variable they align with: continuous, discrete, datetime, or date. There are lots of non-default scales which you’ll learn about below.

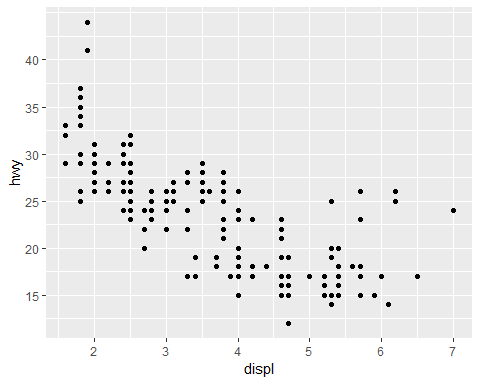
The default scales have been carefully chosen to do a good job for a wide range of inputs. Nevertheless, you might want to override the defaults for two reasons:

* You might want to tweak some of the parameters of the default scale. This allows you to do things like change the breaks on the axes, or the key labels on the legend.
* You might want to replace the scale altogether, and use a completely different algorithm. Often you can do better than the default because you know more about the data.

### 28.4.1 Axis ticks and legend keys

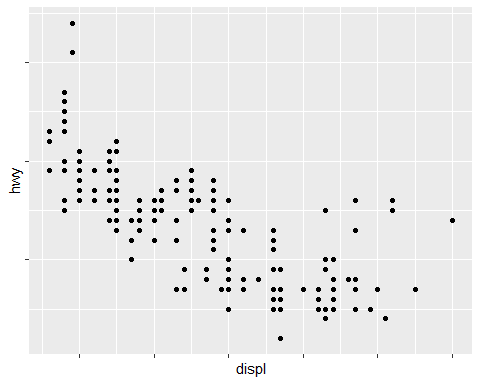
There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: breaks and labels. Breaks controls the position of the ticks, or the values associated with the keys. Labels controls the text label associated with each tick/key. The most common use of breaks is to override the default choice:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point()+  
 scale\_y\_continuous(breaks=seq(15,40,by=5))



You can use labels in the same way (a character vector the same length as breaks), but you can also set it to NULL to suppress the labels altogether. This is useful for maps, or for publishing plots where you can’t share the absolute numbers.

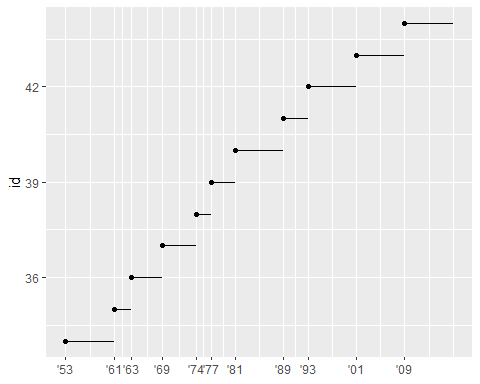
ggplot(mpg,aes(displ,hwy))+  
 geom\_point()+  
 scale\_x\_continuous(labels=NULL)+  
 scale\_y\_continuous(labels = NULL)



You can also use breaks and labels to control the appearance of legends.Collectively axes and legends are called \*\**guides*. Axes are used for x and y aesthetics;legends are used for everything else.

Another use of breaks is when you have relatively few data points and want to highlight exactly where the observations occur. For example, take this plot that shows when each US president started and ended their term.

presidential %>%   
 mutate(id=33+row\_number()) %>%   
 ggplot(aes(start,id))+  
 geom\_point()+  
 geom\_segment(aes(xend=end,yend=id))+  
 scale\_x\_date(NULL,breaks=presidential$start,date\_labels = "'%y")



Note that the specification of breaks and labels for date and datetime scales is a little different:

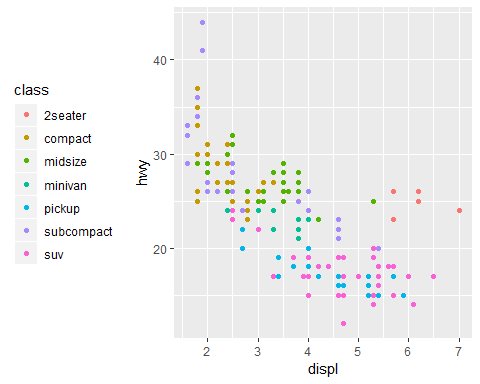
* date\_labels takes a format specification, in the same form as parse\_datetime().
* date\_breaks (not shown here), takes a string like “2 days” or “1 month”.

### 28.4.2 Legend layout

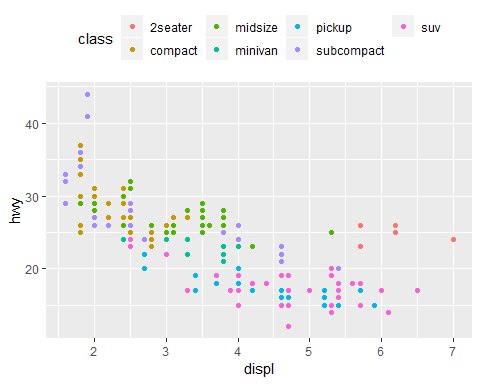
You will most often use breaks and labels to tweak the axes. While they both also work for legends, there are a few other techniques you are more likely to use.

To control the overall position of the legend, you need to use a theme() setting. We’ll come back to themes at the end of the chapter, but in brief, they control the non-data parts of the plot. The theme setting legend.position controls where the legend is drawn:

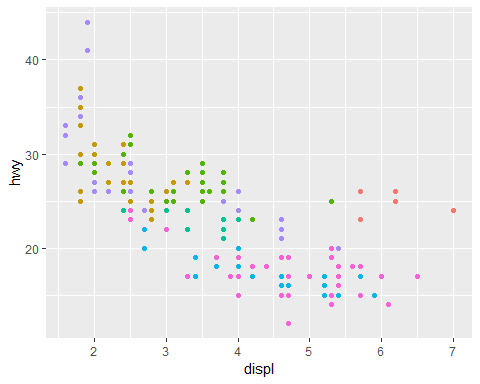
base <- ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))  
  
base + theme(legend.position = "left")



base+theme(legend.position = "top")

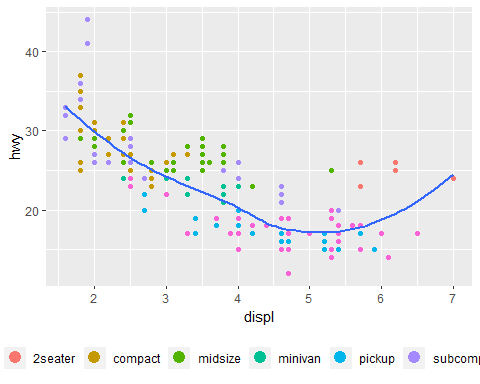


base+theme(legend.position = "")



To control the display of individual legends, use guides() along with guide\_legend() or guide\_colourbar(). The following example shows two important settings: controlling the number of rows the legend uses with nrow, and overriding one of the aesthetics to make the points bigger. This is particularly useful if you have used a low alpha to display many points on a plot.

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(colour=class))+  
 geom\_smooth(se=FALSE)+  
 theme(legend.position="bottom")+  
 guides(colour=guide\_legend(nrow=1,override.aes = list(size=4)))  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



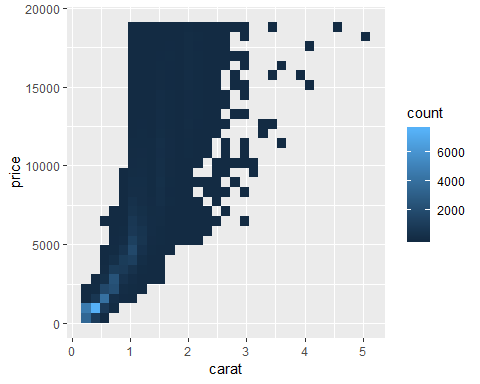
### 28.4.3 Replacing a scale

Instead of just tweaking the details a little, you can instead replace the scale altogether. There are two types of scales you’re most likely to switch out: continuous position scales and colour scales.

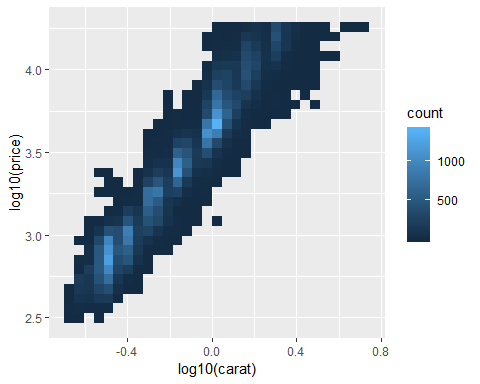
Fortunately, the same principles apply to all the other aesthetics, so once you’ve mastered position and colour, you’ll be able to quickly pick up other scale replacements.

It’s very useful to plot transformations of your variable. For example, as we’ve seens in diamond prices, it’s easier to see the precise relationship between carat and price if we log transform them:

ggplot(diamonds,aes(carat,price))+  
 geom\_bin2d()

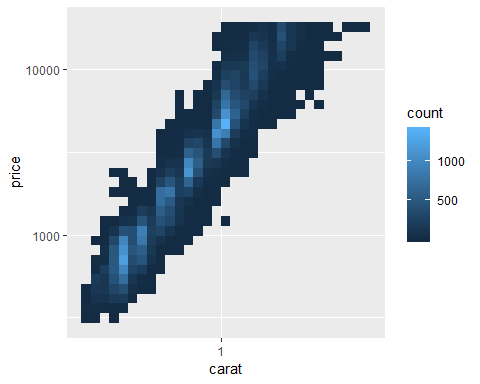


# log transformation  
ggplot(diamonds,aes(log10(carat),log10(price)))+  
 geom\_bin2d()



However, the disadvantage of this transformation is that the axes are now labelled with the transformed values, making it hard to interpret the plot. Instead of doing the transformation in the aesthetic mapping, we can instead do it with the scale. This is visually identical, except the axes are labelled on the original data scale.

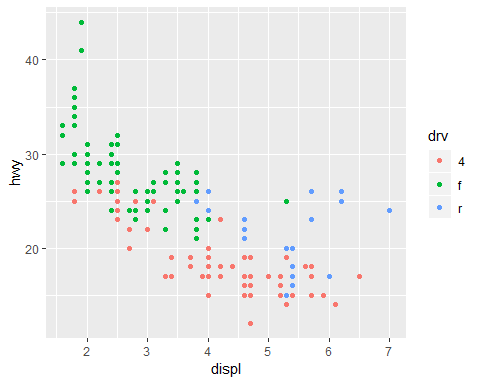
ggplot(diamonds,aes(carat,price))+  
 geom\_bin2d()+  
 scale\_x\_log10()+  
 scale\_y\_log10()



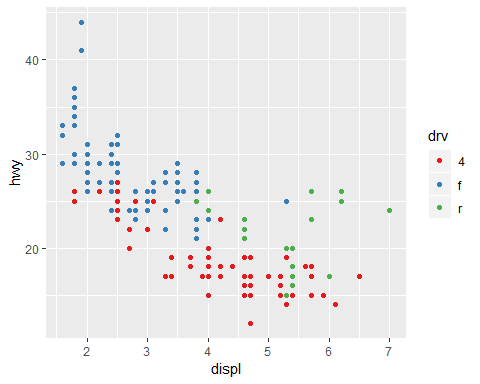
Another scale that is frequently customised is colour. The default categorical scale picks colours that are evenly spaced around the colour wheel. Useful alternatives are the ColorBrewer scales which have been hand tuned to work better for people with common types of colour blindness.

The two plots below look similar, but there is enough difference in the shades of red and green that the dots on the right can be distinguished even by people with red-green colour blindness.

ggplot(mpg,aes(displ,hwy))+  
 geom\_point((aes(color=drv)))

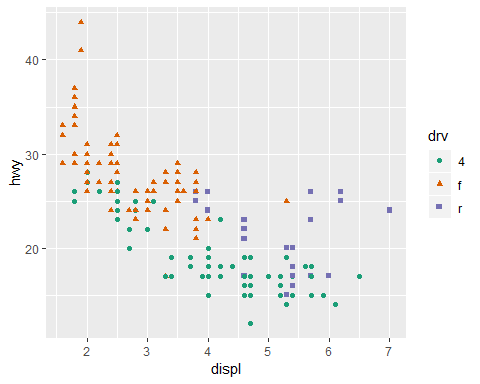


ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=drv))+  
 scale\_colour\_brewer(palette="Set1")



Don’t forget simpler techniques. If there are just a few colours, you can add a redundant shape mapping. This will also help ensure your plot is interpretable in black and white.

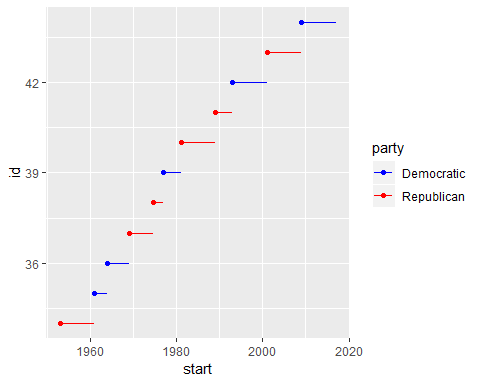
ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=drv,shape=drv))+  
 scale\_color\_brewer(palette="Dark2")



The ColorBrewer scales are documented online at <http://colorbrewer2.org/> and made available in R via the **RColorBrewer** package, by Erich Neuwirth. Figure 28.2 shows the complete list of all palettes. The sequential (top) and diverging (bottom) palettes are particularly useful if your categorical values are ordered, or have a “middle”. This often arises if you’ve used cut() to make a continuous variable into a categorical variable.

When you have a predefined mapping between values and colours, use scale\_colour\_manual(). For example, if we map presidential party to colour, we want to use the standard mapping of red for Republicans and blue for Democrats:

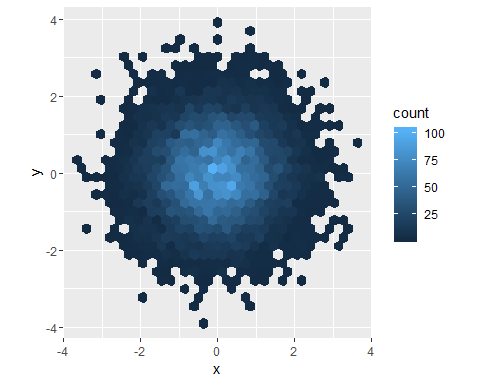
head(presidential)  
#> # A tibble: 6 x 4  
#> name start end party   
#> <chr> <date> <date> <chr>   
#> 1 Eisenhower 1953-01-20 1961-01-20 Republican  
#> 2 Kennedy 1961-01-20 1963-11-22 Democratic  
#> 3 Johnson 1963-11-22 1969-01-20 Democratic  
#> 4 Nixon 1969-01-20 1974-08-09 Republican  
#> 5 Ford 1974-08-09 1977-01-20 Republican  
#> 6 Carter 1977-01-20 1981-01-20 Democratic  
  
presidential %>%   
 mutate(id=33+row\_number()) %>%   
 ggplot(aes(start,id,colour=party))+  
 geom\_point()+  
 geom\_segment(aes(xend=end,yend=id))+  
 scale\_color\_manual(values=c(Republican="red",Democratic="blue"))



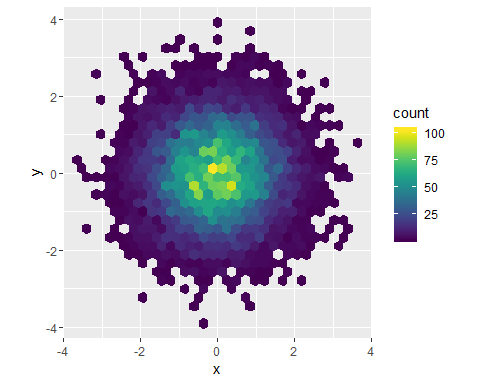
For continuous colour, you can use the built-in scale\_colour\_gradient() or scale\_fill\_gradient(). If you have a diverging scale, you can use scale\_colour\_gradient2(). That allows you to give, for example, positive and negative values different colours. That’s sometimes also useful if you want to distinguish points above or below the mean.

Another option is scale\_colour\_viridis() provided by the **viridis** package. It’s a continuous analog of the categorical ColorBrewer scales. The designers, Nathaniel Smith and Stéfan van der Walt, carefully tailored a continuous colour scheme that has good perceptual properties. Here’s an example from the **viridis** vignette.

df <- tibble(  
 x=rnorm(10000),  
 y=rnorm(10000)  
)  
  
ggplot(df,aes(x,y))+  
 geom\_hex()+  
 coord\_fixed()  
#> Warning: package 'hexbin' was built under R version 3.5.1



ggplot(df,aes(x,y))+  
 geom\_hex()+  
 viridis::scale\_fill\_viridis()+  
 coord\_fixed()

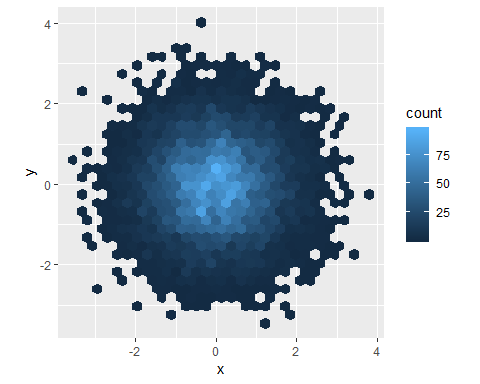


Note that all colour scales come in two variety: scale\_colour\_x() and scale\_fill\_x() for the colour and fill aesthetics respectively (the colour scales are available in both UK and US spellings).

### 28.4.4 Exercises

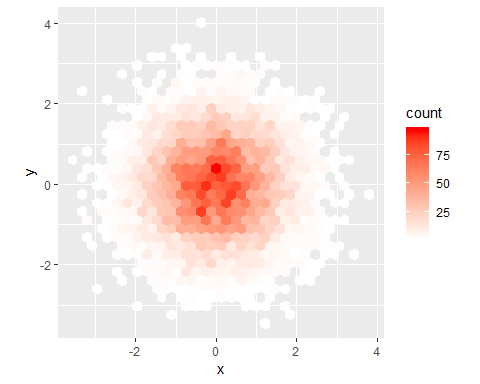
1. Why doesn’t the following code override the default scale?

df <- tibble(  
 x = rnorm(10000),  
 y = rnorm(10000)  
)  
  
ggplot(df, aes(x, y)) +  
 geom\_hex() +  
 scale\_colour\_gradient2(low = "white", high = "red") +  
 coord\_fixed()



I think it’s because there’s not color aesthetic. Instead, geom\_hex uses a fill aesthetic.

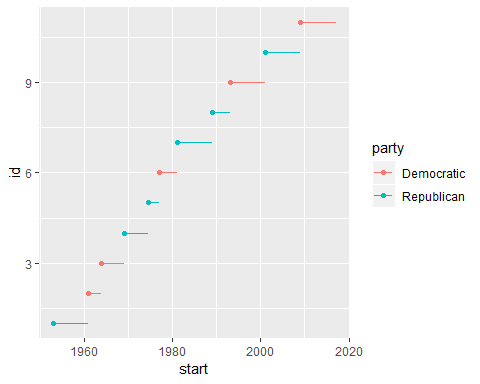
df %>%   
 ggplot(aes(x,y))+  
 geom\_hex()+  
 scale\_fill\_gradient(low="white",high="red")+  
 coord\_fixed()



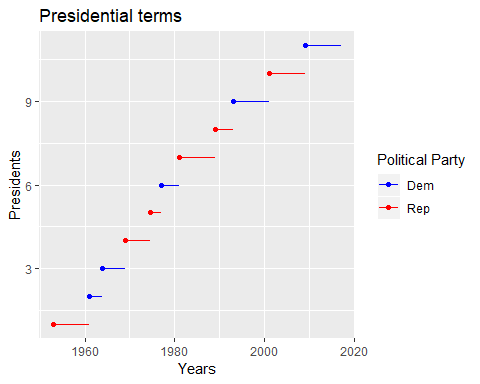
**2. What is the first argument to every scale? How does it comare to labs()?**

**3. Change the display of the presidential terms by**: - Combine the two variants shown above. - Improve the display of y axis. - Labelling each term with the name of the president. - Adding informative plot labels. - Placing breaks every 4 years

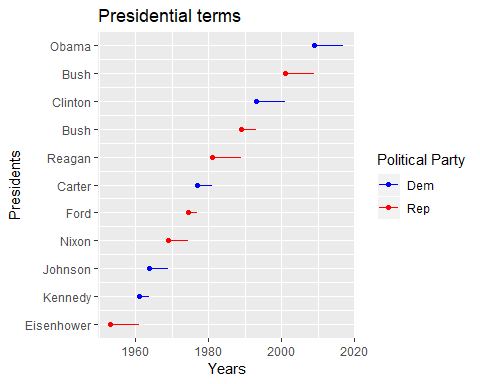
head(presidential)  
#> # A tibble: 6 x 4  
#> name start end party   
#> <chr> <date> <date> <chr>   
#> 1 Eisenhower 1953-01-20 1961-01-20 Republican  
#> 2 Kennedy 1961-01-20 1963-11-22 Democratic  
#> 3 Johnson 1963-11-22 1969-01-20 Democratic  
#> 4 Nixon 1969-01-20 1974-08-09 Republican  
#> 5 Ford 1974-08-09 1977-01-20 Republican  
#> 6 Carter 1977-01-20 1981-01-20 Democratic  
  
presidential <-   
 presidential %>%   
 mutate(id=row\_number())  
  
p <-   
 ggplot(presidential,aes(start,id,color=party))+  
 geom\_point()+  
 geom\_segment(aes(xend=end,yend=id))  
p



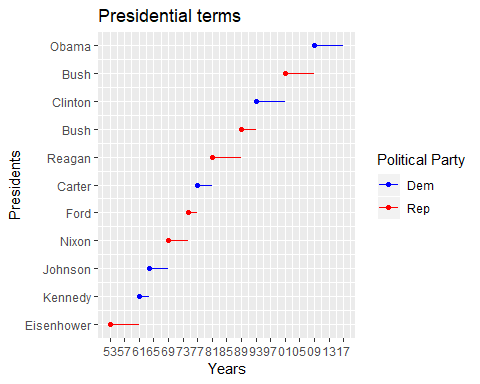
# combining the two variants shown above.  
p2 <-   
 p+  
 labs(title="Presidential terms",  
 x="Years",  
 y="Presidents",  
 color="Political Party")+  
 scale\_color\_manual(labels=c("Dem","Rep"),  
 values=c(Democratic="blue",  
 Republican="red"))  
p2



# Improving the display of the y axis. Labeling each term with the name of the president.  
p3 <- p2+  
 scale\_y\_continuous(breaks=presidential$id,  
 labels=presidential$name)  
p3

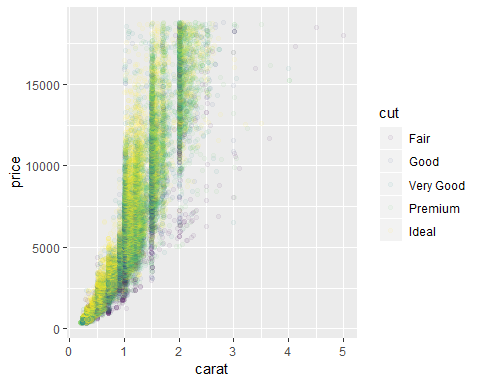


# placing breaks every 4 years  
p4 <- p3+  
 scale\_x\_date(date\_breaks = "4 years",  
 date\_labels="%y")  
p4

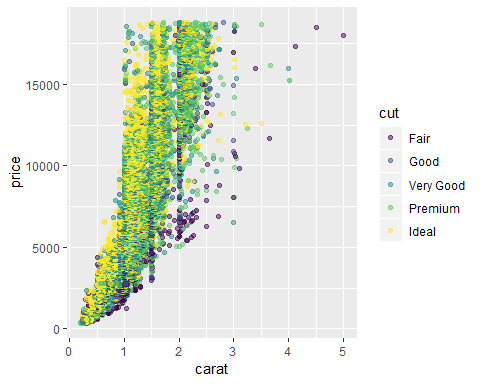


**4. Use override.aes to make the legend of the following plot easier to see.**

head(diamonds)  
#> # A tibble: 6 x 10  
#> carat cut color clarity depth table price x y z  
#> <dbl> <ord> <ord> <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>  
#> 1 0.23 Ideal E SI2 61.5 55 326 3.95 3.98 2.43  
#> 2 0.21 Premium E SI1 59.8 61 326 3.89 3.84 2.31  
#> 3 0.23 Good E VS1 56.9 65 327 4.05 4.07 2.31  
#> 4 0.290 Premium I VS2 62.4 58 334 4.2 4.23 2.63  
#> 5 0.31 Good J SI2 63.3 58 335 4.34 4.35 2.75  
#> 6 0.24 Very Good J VVS2 62.8 57 336 3.94 3.96 2.48  
diamonds %>%   
 distinct(color)  
#> # A tibble: 7 x 1  
#> color  
#> <ord>  
#> 1 E   
#> 2 I   
#> 3 J   
#> 4 H   
#> 5 F   
#> 6 G   
#> 7 D  
  
ggplot(diamonds,aes(carat,price))+  
 geom\_point(aes(colour=cut),alpha=1/20)



# solution  
ggplot(diamonds, aes(carat, price)) +  
 geom\_point(aes(colour = cut), alpha = 0.5) +  
 guides(  
 override.aes = list(alpha = 1)  
 )

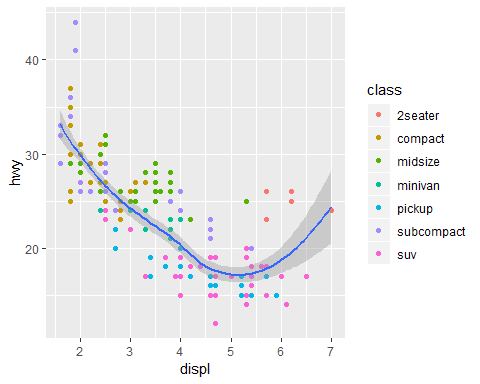


## 28.5 Zooming

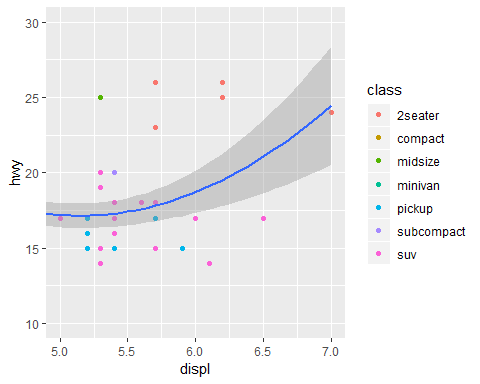
There are three ways to control the plot limits: 1. Adjusting what data are plotted 2. Setting the limits in each scale 3. Setting xlim and ylim in coord\_cartesian()

To zoom in on a region of the plot, iy’s generally best to use coord\_cartesian(). Compare the following two plots:

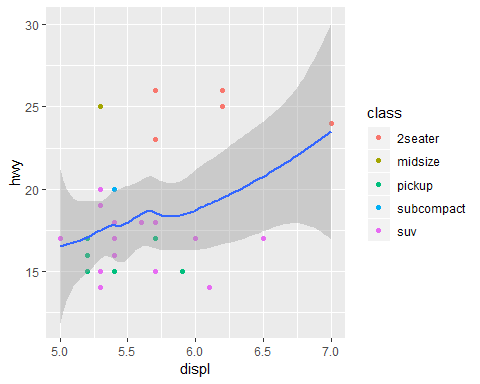
ggplot(mpg,mapping=aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth()  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



ggplot(mpg,mapping=aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth()+  
 coord\_cartesian(xlim=c(5,7),ylim=c(10,30))  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



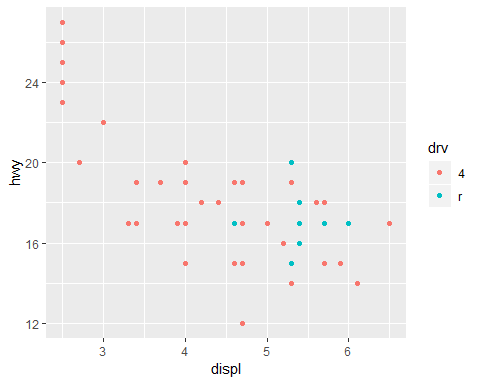
mpg %>%  
 filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%  
 ggplot(aes(displ, hwy)) +  
 geom\_point(aes(color = class)) +  
 geom\_smooth()  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



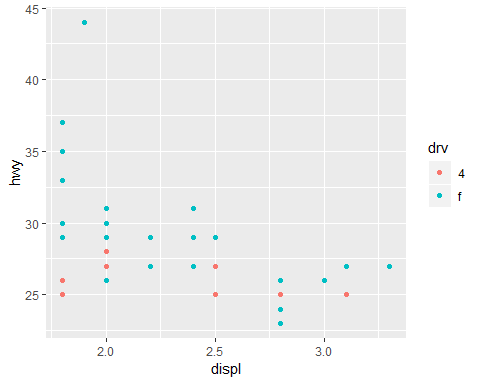
You can also see the limits on individual scales. Reducing the limits is basically equivalent to **subsetting** the data.

It is generally more useful if you want **expand** the limits, for example, to match scales across different plots. For example, if we extract two classes of cars and plot them separately, it’s difficult to compare the plots because all three scales (the x-axis, the y-axis, and the colour aesthetic) have different ranges.

suv <- mpg %>% filter(class=="suv")  
compact <- mpg %>% filter(class=="compact")  
  
ggplot(suv,aes(displ,hwy,colour=drv))+  
 geom\_point()

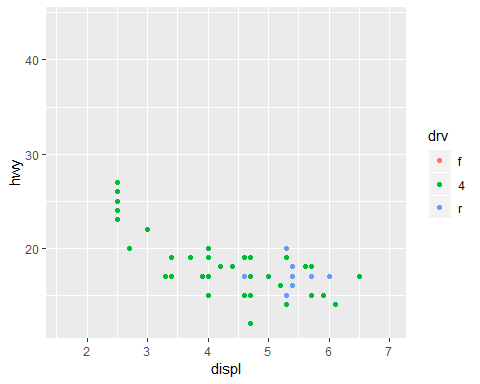


ggplot(compact,aes(displ,hwy,colour=drv))+  
 geom\_point()

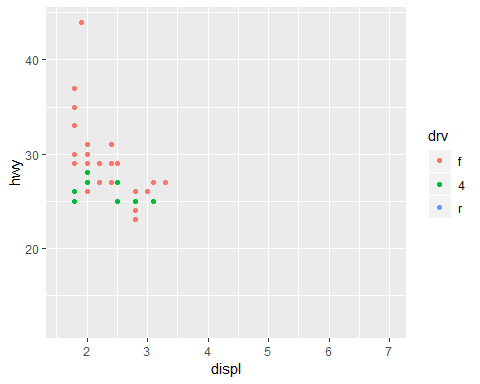


One way to overcome this problem is to share scales across multiple plots, training the scales with the limits of the full data.

x\_scale <- scale\_x\_continuous(limits=range(mpg$displ))  
y\_scale <- scale\_y\_continuous(limits=range(mpg$hwy))  
col\_scale <- scale\_colour\_discrete(limits=unique(mpg$drv))  
  
ggplot(suv, aes(displ, hwy, colour = drv)) +  
 geom\_point() +  
 x\_scale +  
 y\_scale +  
 col\_scale



ggplot(compact, aes(displ, hwy, colour = drv)) +  
 geom\_point() +  
 x\_scale +  
 y\_scale +  
 col\_scale

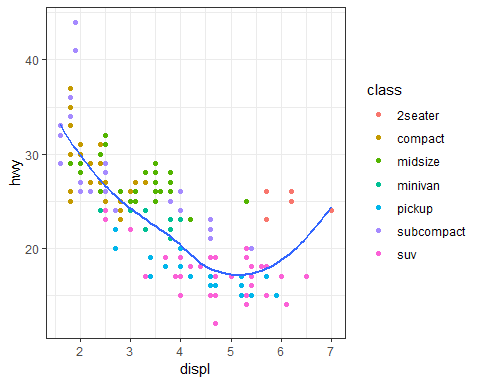


In this particular case, you could have simply used faceting, but this technique is useful more generally, if for instance, you want spread plots over multiple pages of a report.

## 28.6 Themes

Finally, you can customize the non-data elements of your plot with a theme:

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth(se=FALSE)+  
 theme\_bw()  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'

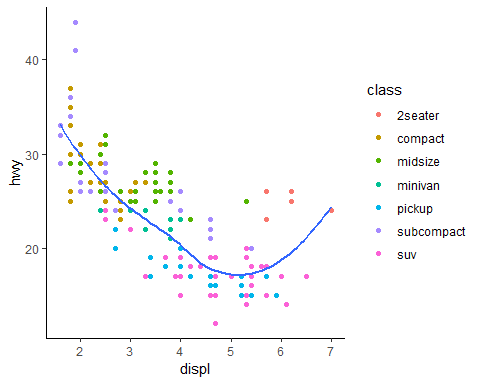


ggplot2 includes eight themes by default, as shown in Figure 29.3.

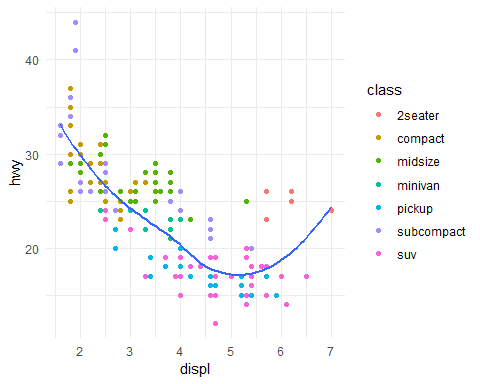
1. theme\_bw()
2. theme\_classic()
3. theme\_dark()
4. theme\_gray()
5. theme\_light()
6. theme\_linedraw()
7. theme\_minimal()
8. theme\_void()

Many more are included in add-on packages like ggthemes(<https://github.com/jrnold/ggthemes>).

ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth(se=FALSE)+  
 # theme\_minimal()  
 theme\_classic()  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



ggplot(mpg,aes(displ,hwy))+  
 geom\_point(aes(color=class))+  
 geom\_smooth(se=FALSE)+  
 # theme\_minimal()  
 theme\_minimal()  
#> `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



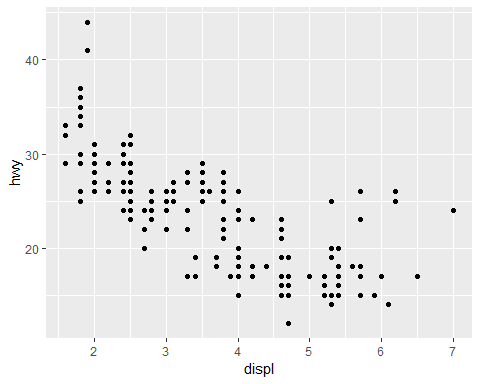
Many people wonder why the default theme has a grey background. This was a deliberate choice because it puts the data forward while still making the grid lines visible. The white grid lines are visible (which is important because they significantly aid position judgements), but they have little visual impact and we can easily tune them out. The grey background gives the plot a similar typographic colour to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the grey background creates a continuous field of colour which ensures that the plot is perceived as a single visual entity.

It’s also possible to control individual components of each theme, like the size and colour of the font used for the y axis. Unfortunately, this level of detail is outside the scope of this book, so you’ll need to read the ggplot2 book for the full details. You can also create your own themes, if you are trying to match a particular corporate or journal style.

## 28.7 Saving your plots

There are two main ways to get your plots out of R and into your final write-up: ggsave() and knitr. ggsave() will save the most recent plot to disk:

ggplot(mpg,aes(displ,hwy))+geom\_point()



ggsave("my-plot.pdf")  
#> Saving 5 x 4 in image

Generally, however, I think you should be assembling your final reports using R Markdown, so I want to focus on the important code chunk options that you should know about for graphics. You can learn more about ggsave() in the documentation.

### 28.7.1 Figure sizing

The biggest challenge of graphics in R Markdown is getting your figures the right size and shape. There are five main options that control figure sizing: fig.width, fig.height, fig.asp, out.width and out.height. Image sizing is challenging because there are two sizes (the size of the figure created by R and the size at which it is inserted in the output document), and multiple ways of specifying the size (i.e., height, width, and aspect ratio: pick two of three).

I only use three of the five options: - I find it most aesthetically pleasing for plots to have a consistent width. To enforce this, I set fig.width = 6 (6“) and fig.asp = 0.618 (the golden ratio) in the defaults. Then in individual chunks, I only adjust fig.asp.

* I control the output size with out.width and set it to a percentage of the line width). I default to out.width = "70%" and fig.align = "center". That give plots room to breathe, without taking up too much space.
* To put multiple plots in a single row I set the out.width to 50% for two plots, 33% for 3 plots, or 25% to 4 plots, and set fig.align = "default". Depending on what I’m trying to illustrate ( e.g. show data or show plot variations), I’ll also tweak fig.width, as discussed below.

If you find that you’re having to squint to read the text in your plot, you need to tweak fig.width. If fig.width is larger than the size the figure is rendered in the final doc, the text will be too small; if fig.width is smaller, the text will be too big. You’ll often need to do a little experimentation to figure out the right ratio between the fig.width and the eventual width in your document. To illustrate the principle, the following three plots have fig.width of 4, 6, and 8 respectively:

If you want to make sure the font size is consistent across all your figures, whenever you set out.width, you’ll also need to adjust fig.width to maintain the same ratio with your default out.width. For example, if your default fig.width is 6 and out.width is 0.7, when you set out.width = "50%" you’ll need to set fig.width to 4.3 (6 \* 0.5 / 0.7).

### 28.7.2 Other important options

When mingling code and text, like I do in this book, I recommend setting fig.show = “hold” so that plots are shown after the code. This has the pleasant side effect of forcing you to break up large blocks of code with their explanations.

To add a caption to the plot, use fig.cap. In R Markdown this will change the figure from inline to “floating”.

If you’re producing PDF output, the default graphics type is PDF. This is a good default because PDFs are high quality vector graphics. However, they can produce very large and slow plots if you are displaying thousands of points. In that case, set dev = “png” to force the use of PNGs. They are slightly lower quality, but will be much more compact.

It’s a good idea to name code chunks that produce figures, even if you don’t routinely label other chunks. The chunk label is used to generate the file name of the graphic on disk, so naming your chunks makes it much easier to pick out plots and reuse in other circumstances (i.e. if you want to quickly drop a single plot into an email or a tweet).

## 28.8. Learning more

The absolute best place to learn more is the [ggplot2 book: ggplot2: Elegant graphics for data analysis](https://www.amazon.com/dp/331924275X/ref=cm_sw_su_dp). It goes into much more depth about the underlying theory, and has many more examples of how to combine the individual pieces to solve practical problems. Unfortunately, the book is not available online for free, although you can find the source code at <https://github.com/hadley/ggplot2-book>.

Another great resource is the ggplot2 extensions guide <http://www.ggplot2-exts.org/>. This site lists many of the packages that extend ggplot2 with new geoms and scales. It’s a great place to start if you’re trying to do something that seems hard with ggplot2.

# Chapter 29: R Markdown formats

## 29.1 Introduction

So far you’ve seen R Markdown used to produce HTML documents. This chapter gives a brief overview of some of the many other types of output you can produce with R Markdown. There are two ways to set the output of a document:

1. Permanently, by modifying the YAML header:

title:"Viridis Demo"  
output: html\_document

1. Transiently, by calling rmarkdown::render() by hand:

rmarkdown::render("diamond-sizes.Rmd",output\_format="word\_document")

This is useful if you want to programmatically produce multiple types of output.RStudio’s knit button renders a file to the first format listed in its output field. You can render to additional formats by clicking the dropdown menu beside the knit button.

## 29.2 output options

Each output format is associated with an R function. You can either write foo or pkg::foo. If you omit pkg, the default is assumed to be rmarkdown. It’s important to know the name of the function that makes the output because that’s where you get help. For example, to figure out what parameters you can set with html\_document, look at ?rmarkdown::html\_document.

To override the default parameter values, you need to use an expanded output field. For example, if you wanted to render an html\_document with a floating table of contents, you’d use:

output:  
 html\_document:  
 toc: true  
 toc\_float: true

You can even render to multiple outputs by supplying a list of formats:

output:  
 html\_document:  
 toc: true  
 toc\_float: true  
 pdf\_document: default

Note the special syntax if you don’t want to override any of the default options.

## 29.3 Documents

The previous chapter focused on the default html\_document output. There are a number of basic variations on that theme, generating different types of documents:

* pdf\_document makes a PDF with LaTeX (an open source document layout system), which you’ll need to install. RStudio will prompt you if you don’t already have it.
* word\_document for Microsoft Word documents (.docx).
* odt\_document for OpenDocument Text documents (.odt).
* rtf\_document for Rich Text Format (.rtf) documents.
* md\_document for a Markdown document. This isn’t typically useful by itself, but you might use it if, for example, your corporate CMS or lab wiki uses markdown.
* github\_document: this is a tailored version of md\_document designed for sharing on GitHub.

Remember, when generating a document to share with decision makers, you can turn off the default display of code by setting global options in the setup chunk:

knitr::opts\_chunk$set(echo=FALSE)

For html\_documents another option is to make the code chunks hidden by default, but visible with a click:

output:  
 html\_document:  
 code\_folding: hide

## 29.4 Notebook

A notebook, html\_notebook, is a variation on a html\_document. The rendered outputs are very similar, but the purpose is different. A html\_document is focused on communicating with decision makers, while a notebook is focused on collaborating with other data scientists. These different purposes lead to using the HTML output in different ways. Both HTML outputs will contain the fully rendered output, but the notebook also contains the full source code. That means you can use the .nb.html generated by the notebook in two ways:

1. You can view it in a web browser, and see the rendered output. Unlike html\_document, this rendering always includes an embedded copy of the source code that generated it.
2. You can edit it in RStudio. When you open an .nb.html file, RStudio will automatically recreate the .Rmd file that generated it. In the future, you will also be able to include supporting files (e.g. .csv data files), which will be automatically extracted when needed.

Emailing .nb.html files is a simple way to share analyses with your colleagues. But things will get painful as soon as they want to make changes. If this starts to happen, it’s a good time to learn Git and Github. Learning Git and GitHub is definitely painful t first, but the collaboration payoff is huge. As mentioned earlier, Git and GitHub are outside th scope of the book, but there’s one tip that’s useful if you are already using them: use both html\_notebook and github\_document outputs:

output:  
 html\_notebook:default  
 github\_document:default

html\_notebook gives you a local preview, and a file that you can share via email. github\_document creates a minimal md file that you can check into git. You can easily see how the results of your analysis (not just the code) change over time, and GitHub will render it for you nicely online.

## 29.5 Presentation

You can also use R Markdown to produce presentations. You get less visual control than with a tool like Keynote or PowerPoint, but automatically inserting the results of your R code into a presentation can save a huge amount of time. Presentations work by dividing your content into slides, with a new slide beginning at each first (#) or second (##) level header. You can also insert a horizontal rule (\*\*\*) to create a new slide without a header.

R Markdown comes with three presenttion fromats built-in:

1.ioslides\_presentation - HTML presentation with ioslides 2. slidy\_presentation - HTML presentation with W3C slidy 3. beamer-presnetation - PDF presentation with LaTeX Beamer

Two other popular fomats are provided by packages:

1. evealjs::revealjs\_presentation - HTML presentation with reveal.js. Requires the revealjs package.
2. **rmdshower** - <https://github.com/MangoTheCat/rmdshower>, provides a wrapper around the shower, <https://github.com/shower/shower>, presentation engine

## 29.6 Dashboards

Dashboards are a useful way to communicate large amounts of information visually and quickly. Flexdashboard makes it particularly easy to create dashboards using R Markdown and a convention for how the headers affect the layout:

* Each level 1 header (#) begins a new page in the dashboard.
* Each level 2 header (##) begins a new column.
* Each level 3 header (###) begins a new row.

For example, you can produce this dashboard using this code:

Flexdashboard also provides simple tools for creating sidebars, tabsets, value boxes, and gauges. To learn more about flexdashboard visit <http://rmarkdown.rstudio.com/flexdashboard/>.

## 29.7 Interactivity

Any HTML format (document, notebook, presentation, or dashboard) can contain interactive components.

### 29.7.1 htmlwidgets

HTML is an interactive format, and you can take advantage of that interactivity with **htmlwidgets**, R functions that produce interactive HTML visualisations. For example, take the leaflet map below. If you’re viewing this page on the web, you can drag the map around, zoom in and out, etc. You obviously can’t do that in a book, so rmarkdown automatically inserts a static screenshot for you.

library(leaflet)  
#> Warning: package 'leaflet' was built under R version 3.5.1  
leaflet() %>%   
 setView(174.764,-36.877,zoom=16) %>%   
 addTiles() %>%   
 addMarkers(174.764,-36.877,popup = "Maungwhau")  
#> PhantomJS not found. You can install it with webshot::install\_phantomjs(). If it is installed, please make sure the phantomjs executable can be found via the PATH variable.

The grant thing about htmlwidgets is that you don’t need to know anything about HTML or JavaScript to use them. All the details are wrapped inside the package, so you don’t need to worry about it.

There are many packages that provide htmlwidgets, including:

* **dygraphs**, <http://rstudio.github.io/dygraphs/>, for interactive time series visualisations.
* **DT**, <http://rstudio.github.io/DT/>, for interactive tables.
* **threejs**, <https://github.com/bwlewis/rthreejs> for interactive 3d plots.
* **DiagrammeR**, <http://rich-iannone.github.io/DiagrammeR/> for diagrams (like flow charts and simple node-link diagrams).

To learn more about htmlwidgets and see more complete list of packages that provide them visit <http://www.htmlwidgets.org/>.

### 29.7.2 Shiny

htmlwidgets provide **client-side** interactivity — all the interactivity happens in the browser, independently of R. On one hand, that’s great because you can distribute the HTML file without any connection to R. However, that fundamentally limits what you can do to things that have been implemented in HTML and JavaScript. An alternative approach is to use shiny, a package that allows you to create interactivity using R code, not JavaScript.

To call Shiny code from an R Markdown document, add runtime: shiny to the header:

title: "Shiny Web App"  
output: html\_document  
runtime: shiny

Then you can use the “input” functions to add interactive components to the document:

library(shiny)  
  
textInput("name", "What is your name?")  
numericInput("age", "How old are you?", NA, min = 0, max = 150)

You can then refer to the values with input$name and input$age, and the code that uses them will be automatically re-run whenever they change.

I can’t show you a live shiny app here because shiny interactions occur on the **server-side**. This means that you can write interactive apps without knowing JavaScript, but you need a server to run them on. This introduces a logistical issue: Shiny apps need a Shiny server to be run online. When you run shiny apps on your own computer, shiny automatically sets up a shiny server for you, but you need a public facing shiny server if you want to publish this sort of interactivity online. That’s the fundamental trade-off of shiny: you can do anything in a shiny document that you can do in R, but it requires someone to be running R.

Learn more about Shiny at <http://shiny.rstudio.com/>.

## 29.8 Websites

With a little additional infrastructure you can use R Markdown to generate a complete website:

* Put your .Rmd files in a single directory. index.Rmd will become the home page.
* Add a YAML file named \_site.yml provides the navigation for the site. For example:

name: "my-website"  
navbar:  
 title: "My Website"  
 left:  
 - text: "Home"  
 href: index.html  
 - text: "Viridis Colors"  
 href: 1-example.html  
 - text: "Terrain Colors"  
 href: 3-inline.html

Execute rmarkdown::render\_site() to build \_site, a directory of files ready to deploy as a standalone static website, or if you use an RStudio Project for your website directory. RStudio will add a Build tab to the IDE that you can use to build and preview your site.

Read more at <http://rmarkdown.rstudio.com/rmarkdown_websites.html>.

## 29.9 Other formats

Other packages provide even more output formats:

* The **bookdown** package, <https://github.com/rstudio/bookdown>, makes it easy to write books, like this one. To learn more, read Authoring Books with R Markdown, by Yihui Xie, which is, of course, written in bookdown. Visit <http://www.bookdown.org> to see other bookdown books written by the wider R community.
* The **prettydoc** package, <https://github.com/yixuan/prettydoc/>, provides lightweight document formats with a range of attractive themes.
* The **rticles** package, <https://github.com/rstudio/rticles>, compiles a selection of formats tailored for specific scientific journals.

See <http://rmarkdown.rstudio.com/formats.html> for a list of even more formats. You can also create your own by following the instructions at <http://rmarkdown.rstudio.com/developer_custom_formats.html>.

## 29.10 Learning more

To learn more about effective communication in these different formats I recommend the following resources:

* To improve your presentation skills, I recommend [Presentation Patterns](https://www.amazon.com/dp/0321820800/ref=cm_sw_su_dp), by Neal Ford, Matthew McCollough, and Nathaniel Schutta. It provides a set of effective patterns (both low- and high-level) that you can apply to improve your presentations.
* If you give academic talks, I recommend reading [the Leek group guide to giving talks](https://github.com/jtleek/talkguide).
* I haven’t taken it myself, but I’ve heard good things about Matt McGarrity’s online course on public speaking: <https://www.coursera.org/learn/public-speaking>.
* If you are creating a lot of dashboards, make sure to read Stephen Few’s [Information Dashboard Design: The Effective Visual Communication of Data](https://www.amazon.com/dp/0596100167/ref=cm_sw_su_dp). It will help you create dashboards that are truly useful, not just pretty to look at.
* Effectively communicating your ideas often benefits from some knowledge of graphic design. The [Non-Designer’s Design Book](https://www.amazon.com/dp/0133966151/ref=cm_sw_su_dp) is a great place to start.

# Chapter 30: R Markdown workflow

Earlier, we discussed a basic workflow for capturing your R code where you work interactively in the *console*, then capture what works in the *script editor*. R Markdown brings together the console and the script editor, blurring the lines between interactive exploration and long-term code capture.

You can rapidly iterate within a chunk, editing and re-executing with Cmd/Ctrl + Shift + Enter. When you’re happy, you move on and start a new chunk.

R Markdown is also important because it so tightly integrates prose and code. This makes it a great analysis notebook because it lets you develop code and record your thoughts. An analysis notebook shares many of the same goals as a classic lab notebook in the physical sciences. It:

* Records what you did and why you did it. Regardless of how great your memory is, if you don’t record what you do, there will come a time when you have forgotten important details. Write them down so you don’t forget!
* Supports rigorous thinking. You are more likely to come up with a strong analysis if you record your thoughts as you go, and continue to reflect on them. This also saves you time when you eventually write up your analysis to share with others.
* Helps others understand your work. It is rare to do data analysis by yourself, and you’ll often be working as part of a team. A lab notebook helps you share not only what you’ve done, but why you did it with your colleagues or lab mates.

Much of the good advice about using lab notebooks effectively can also be translated to analysis notebooks. I’ve drawn on my own experiences and Colin Purrington’s advice on lab notebooks (<http://colinpurrington.com/tips/lab-notebooks>) to come up with the following tips:

* Ensure each notebook has a descriptive title, an evocative filename, and a first paragraph that briefly describes the aims of the analysis.
* Use the YAML header date field to record the date you started working on the notebook:

Use ISO8601 YYYY-MM-DD format so that’s there no ambiguity. Use it even if you don’t normally write dates that way!

* If you spend a lot of time on an analysis idea and it turns out to be a dead end, don’t delete it! Write up a brief note about why it failed and leave it in the notebook. That will help you avoid going down the same dead end when you come back to the analysis in the future.
* Generally, you’re better off doing data entry outside of R. But if you do need to record a small snippet of data, clearly lay it out using tibble::tribble().
* If you discover an error in a data file, never modify it directly, but instead write code to correct the value. Explain why you made the fix.
* Before you finish for the day, make sure you can knit the notebook (if you’re using caching, make sure to clear the caches). That will let you fix any problems while the code is still fresh in your mind.
* If you want your code to be reproducible in the long-run (i.e. so you can come back to run it next month or next year), you’ll need to track the versions of the packages that your code uses. A rigorous approach is to use **packrat**, <http://rstudio.github.io/packrat/>, which stores packages in your project directory, or **checkpoint**, <https://github.com/RevolutionAnalytics/checkpoint>, which will reinstall packages available on a specified date. A quick and dirty hack is to include a chunk that runs sessionInfo() — that won’t let you easily recreate your packages as they are today, but at least you’ll know what they were.

^ You are going to create many, many, many analysis notebooks over the course of your career. How are you going to organise them so you can find them again in the future? I recommend storing them in individual projects, and coming up with a good naming scheme.