How To Make Mistakes In R

Andrew Flowers

2017-03-10

Contents

Chapter 1

Introduction

1.0.1 R is great, but it's also weird

R was built by and for statisticians, so it's not like other programming languages. Its idiosyncrasies can be a source of deep frustration for beginners. But I'd argue there is no better tool for data analysis.

That's why I've written this free ebook **How To Make Mistakes In R** for O'Reilly. It's modeled after the excellent **How To Make Mistakes In Python**, by Mike Pirnat.

The target audience is all R programmers, from those just starting out all the way to the advanced developers. It'll cover mistakes in setup, silly things, style no-no's, statistics and surprises, too. I'm especially qualified to write this book because I've made so many R mistakes in my own work.

1.0.2 R is great for data science

The focus of this book is *not* on programming in R as much as it is using R for data science. This is where R is at its best: an environment for interactively exploring and analyzing data.

Thus we'll often be working with real datasets. To access those datasets, go to this book's GitHub repository.

1.0.3 What to do when you make a mistake in R?

Making mistakes in programming is perfectly normal. It's to be expected, really, in the course of anyone's workflow. But how should you react when you see an R tells you that you've made an error and you don't understand the error message.

The first step should be **Google the error message**. Yes, that's my advice. As simple as it sounds, many programmers – especially beginners – go hunting for answers by asking others or searching R documentation. They don't go to the source of their problem: the error message.

```
sum(1, 2, 'hello')
```

```
## Error in sum(1, 2, "hello"): invalid 'type' (character) of argument
```

And often by Googling the error message you'll end up at Stackoverflow, the live-saving website where programmers post questions and (sometimes) get answers. Chances are: someone has already made your mistake in R before. And, hopefully, there's an answer at Stackoverflow.

But what if you're not getting an explicit error message from R, but your code isn't working as you expect. A simple place to start is to read R documentation about the functions your using. Just type into the R console your function name preface with a `?` and the function documentation will appear. Like this:

```
?sum
```

After Googling and searching Stackoverflow, and examiming R documentation, other places to find for answers are GitHub and Twitter. The `#rstats` hash tag on Twitter collects lots of useful conversations about R.

1.0.4 Good luck and keep at it.

Mistakes are inevitable in programming. Don't think you're stupid. Don't think your frustration is unusal. Frustration is to be expected. Nothing is wrong with you. You just need to keep at it.

Chapter 2

Setup

Getting started with R can be frustrating. Actually, no, it's *guaranteed* to be frustrating at the beginning But that's entirely normal! Take a deep breath, persevere through the initial confusion and know that it gets easier. I promise that you'll be glad you did.

To make things easier on you, in this chapter we'll cover the common beginner-level mistakes in just getting your R coding environment in order. (I'll assume you've installed R.)

2.0.1 RStudio IDE

First things first: what IDE are using? *"ID-what?"* you might say. An integrated development environment (IDE) is like a workstation. For your physical workstation, you probably prefer an ergonomic chair or maybe a modern standing desk. A good workstation can make a big difference in your productivity. Ditto for your digital workstation – so take care in choosing.

Now, you don't have to use an IDE to write R code; you can execute R commands and write R scripts using the pre-built R framework. But I wouldn't recommend it.

By far the most widely-used IDE for R is RStudio. If you haven't yet, go install Rstudio. RStudio makes writing R code less of a headache.

2.0.2 More than one way to install a package

R is a set of tools. "Base R" – the pre-installed set of functions like `sum()` and `plot()` – is like a starter toolbelt, with just a hammer and wrench. It's the minimal tools you need to accomplish simple tasks.

However, to do anything meaningful in R you need to install other R packages. Specifically, you need to install *and* load R packages into your R library. Perhaps the most common mistake that R newbies make is loading a package *before* installing it:

```
# NOTE: there is no 'RPACKAGE' package

library(RPACKAGE)
```

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

The `library()` function loads an already-installed package into your R environment – the set of packages you've activated at the moment. But for it to work, you first need install it; and most often you do that with the `install.packages()` function. Let's start by installing and loading the `tidyverse` package, which is really a collection of packages, with the package name in quotes like `'tidyverse'`.

The two-step process goes like this:

```
# Step 1: Install the package
install.packages('tidyverse')

# Step 2: load the package
library(tidyverse)

# NOTE: quotes are optional for the library() function
```

You often will use the `install.packages()` function because it taps directly into CRAN, the Comprehensive R Archive Network, where thousands of R packages are approved and maintained. But this is not the only way to install an R package.

Another common method is via GitHub, but to do that you first need another R package called `devtools`. So first we'll install `devtools` the old-fashioned way and then use its nifty function to install a second package – called `bookdown` – directly via GitHub. Just use the `install_github()` funciton with the author's name, a `/` and the package name. Because the GitHub user `rstudio` is the author of the `bookdown` package, it looks like this:

```
# Install and load the devtools package
install.package("devtools")
library(devtools)

# Install the bookdown package (authored by rstudio) from GitHub
install_github("rstudio/bookdown")
```

You could have installed `bookdown` through CRAN and the `install.packages()` function. But the `devtools` package, with its `install_github` function, allows you to broaden what R packages you can use.

2.0.3 Working directory confusion

Another common R mistake is not being aware of your **working directory**, or the file/folder on your computer from which you're currently working. R has useful functions – `getwd()` and `setwd()` – for discovering and changing your working directory. And I'd also recommend the `dir()` function to list which files are within your current working directory. Below I use `getwd()` to list my current working directory and the files within it, and then I use `setwd()` with `..` to move one folder level higher in my directory.

```
getwd()
```

```
## [1] "/Users/andrewflowers/projects/oreilly/mistakes-in-R/draft-mistakes-in-R"
```

```
dir()
```

```
##  [1] "_book"                          "_bookdown_files"
##  [3] "_bookdown.yml"                  "_output.yml"
##  [5] "01-intro.Rmd"                   "02-setup.Rmd"
##  [7] "03-silly.Rmd"                   "04-style.Rmd"
##  [9] "05-stats.Rmd"                   "06-surprises.Rmd"
## [11] "assets"                         "common_names.csv"
## [13] "draft-mistakes-in-R.Rproj"      "how_to_make_mistakes_in_R_cache"
## [15] "how_to_make_mistakes_in_R_files" "how_to_make_mistakes_in_R.Rmd"
## [17] "index.Rmd"                      "nba_elo_data.csv"
## [19] "README.md"
```

```
# To move my working directory one folder higher:
setwd('..')
```

```
getwd()
```

```
## [1] "/Users/andrewflowers/projects/oreilly/mistakes-in-R"
```

If you've ever tried to import a data file – like a `.csv` with data – but get an erorr, it's probably because you've mistunderstood your working directory (or didn't correctly specifcy the path to your data file). See here for instance:

```
data <- read.csv("my-data-file.csv")
```

```
## Warning in file(file, "rt"): cannot open file 'my-data-file.csv': No such
## file or directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

```
# NOTE: my-data-file.csv is not in my current working directory

dir()
```

```
##  [1] "_book"                           "_bookdown_files"
##  [3] "_bookdown.yml"                   "_output.yml"
##  [5] "01-intro.Rmd"                    "02-setup.Rmd"
##  [7] "03-silly.Rmd"                    "04-style.Rmd"
##  [9] "05-stats.Rmd"                    "06-surprises.Rmd"
## [11] "assets"                          "common_names.csv"
## [13] "draft-mistakes-in-R.Rproj"       "how_to_make_mistakes_in_R_cache"
## [15] "how_to_make_mistakes_in_R_files" "how_to_make_mistakes_in_R.Rmd"
## [17] "index.Rmd"                       "nba_elo_data.csv"
## [19] "README.md"
```

2.0.4 RStudio Projects

There is an elegant fix to working directory headaches: RStudio Projects. Projects automatically set your working directory to the location of the project, and provide a neat way to organize all your files (`.R` scripts, data files and so on). To create a project within RStudio, click in the top-right corner.

You'll then choose whether to create a new directory (or folder) to hold the project, or using an existing one.

Most likely, you'll want to create an empty project.

Then you'll specify the project name. Voila! Now you're all set.

2.0.5 *The* biggest R mistake

And that brings up what is probably the biggest R mistake you can make: **not using an R package that suits your problem, or using the wrong package**. Unfortunately, there is no error message for this; you learn it through experience.

For data science problems, though, a good place to start is the `tidyverse` package we just loaded. We'll be referencing its various packages throughout this book. But when in doubt, look there first.

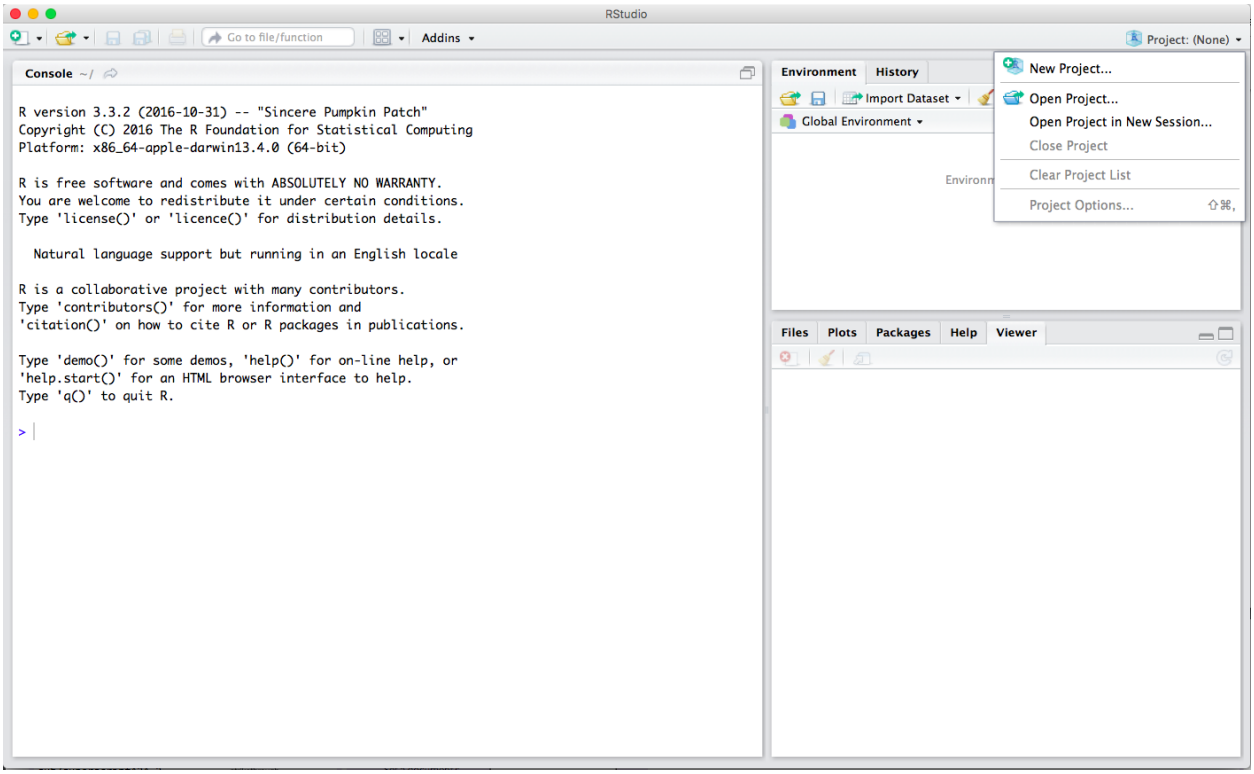

RStudio
Go to file/function
Addins
Project: (None)
Console ~/
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
Environment
History
Import Dataset
Global Environment
New Project...
Open Project...
Open Project in New Session...
Close Project
Clear Project List
Project Options...
Files
Plots
Packages
Help
Viewer

Figure 2.1:

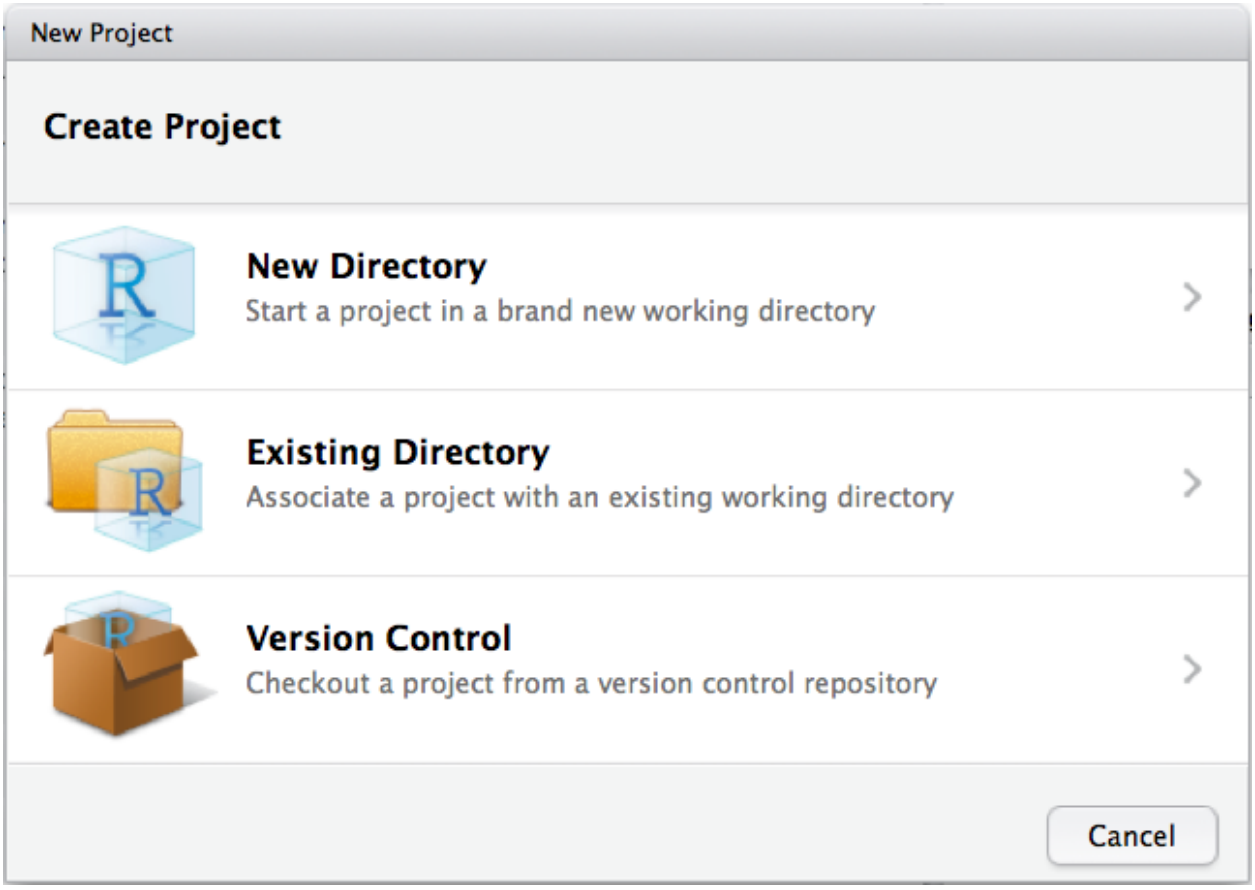

New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

Figure 2.2:

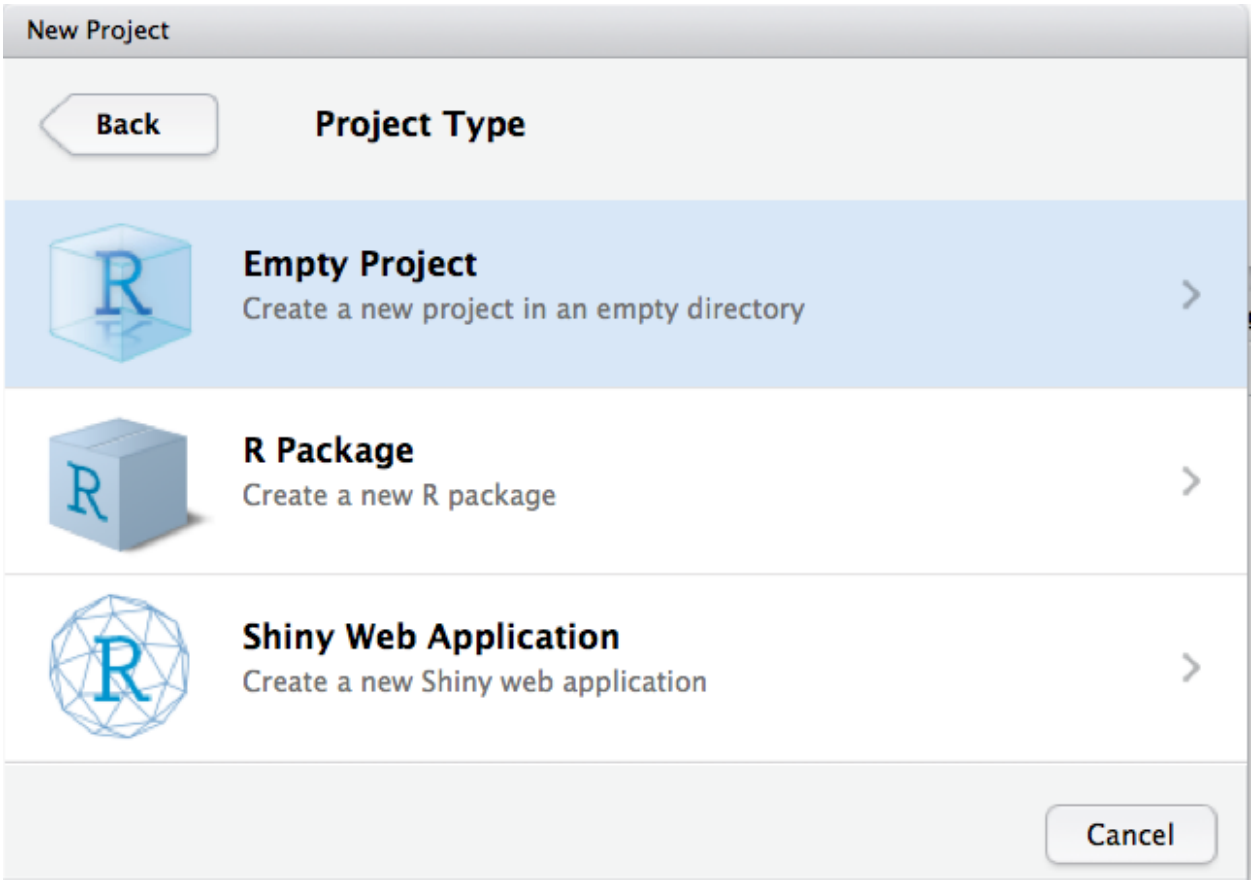

New Project
Back
Project Type
Empty Project
Create a new project in an empty directory
R Package
Create a new R package
Shiny Web Application
Create a new Shiny web application
Cancel

Figure 2.3:

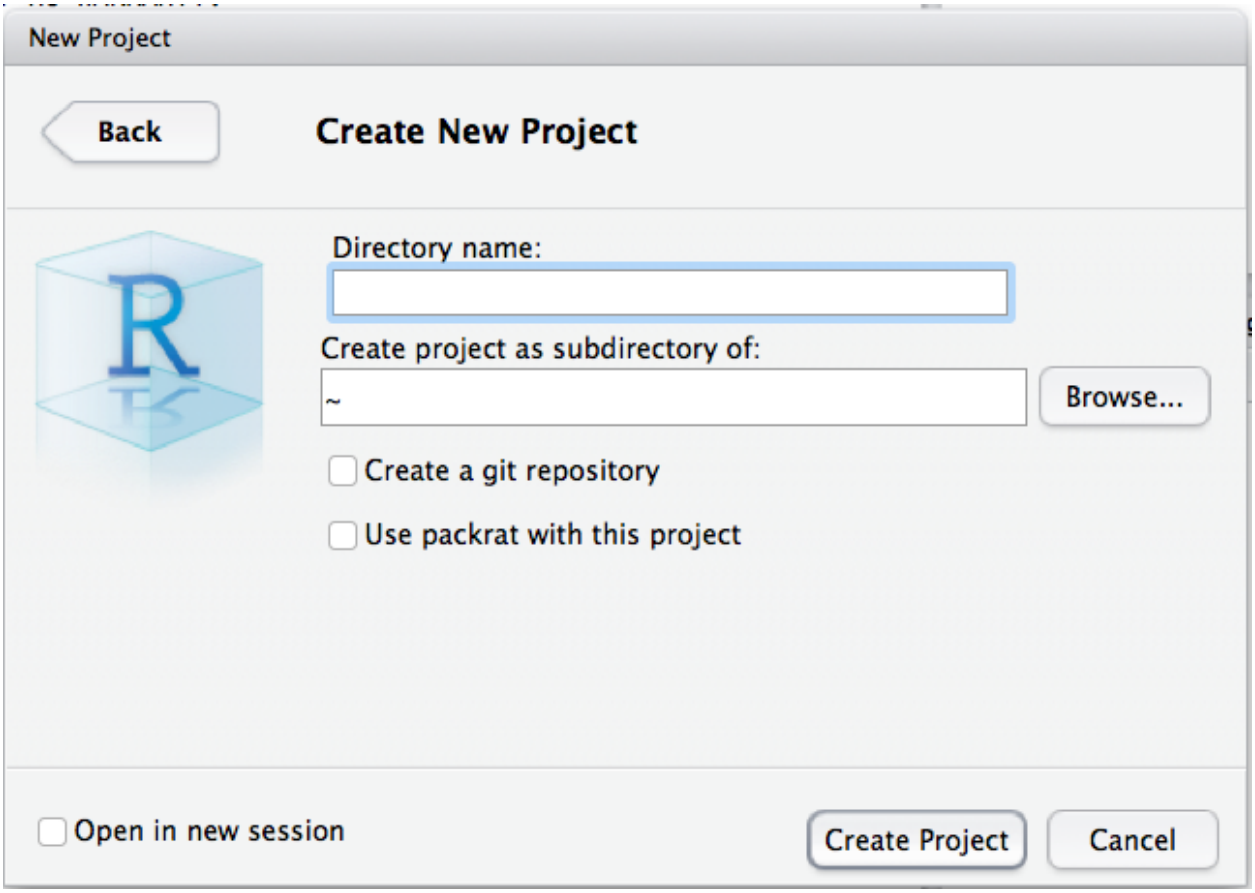

New Project
Back
Create New Project
Directory name:
Create project as subdirectory of:
~
Browse...
Create a git repository
Use packrat with this project
Open in new session
Create Project
Cancel

Figure 2.4:

Chapter 3

Silly things

Now you should be setup for success in R – you're using RStudio, can install packages and are working through RStudio projects. But there will still be lots of frustrating quirks to learn about R. And often your mistakes, you'll come to find, are silly little things. In this chapter we'll cover those common screwups in actual R coding

3.0.1 Basics

R is very picky. Every `(` must be paired with a corresponding `)`. Variable names most be spelled correctly, and when passing a file name or other string to an R function, it must be surrounded by quotes (e.g. `'dir/file-name.csv'`).

Usually, it's best to work in the RStudio Console to interactively run R commands. And while sometimes you'll get an error, other types the console will leave you hanging with an "+" while your cursor is blinking. Like here when I forget the second `)`:

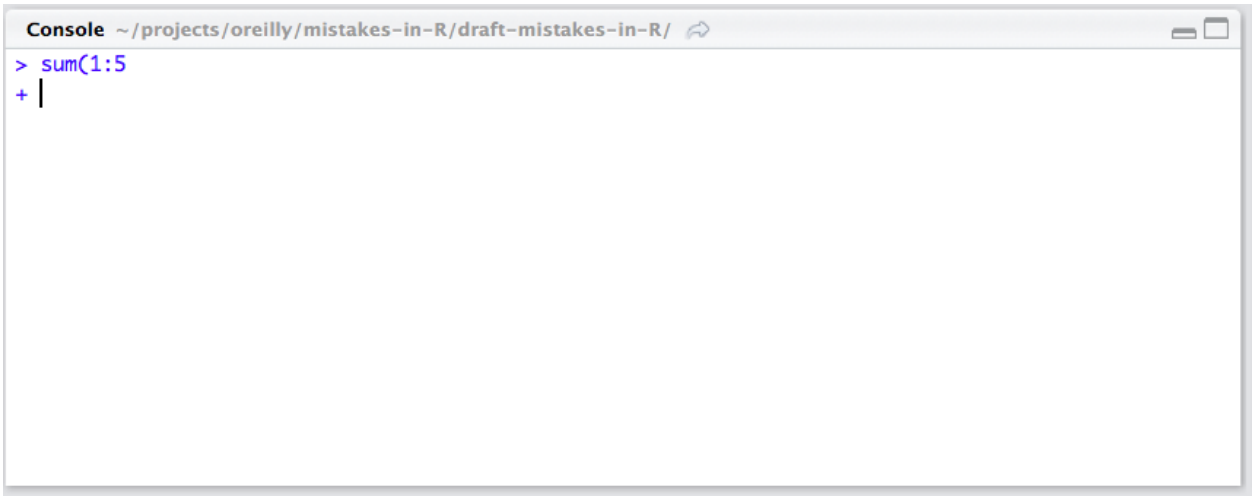


Figure 3.1:

This means your command is incomplete and RStudio is waiting. Just press the "esc" key.

3.0.2 Strings as factors

Every programming language has basic data structures for numbers, strings and booleans. R is no different: there are `double`, `character` and `logical` data types. There is also the `list` for heterogeneous data. But the best known – and most influential – R data structrue is `data.frame`. A dataframe contains tabular data, like a spreadsheet. More precisely, it's a list of equal-length vectors.

However, one annoying default with R is how it treats strings (or character) data in dataframes. Unless you specify otherwise, it interprets this data as a `factor`. A factor can be a useful data structure. A `factor` is a vector with only **pre-defined** values, and so it works well with categorical data (like `male`/`female` or `democrat`/`republican`). Yet often you do *not* want your non-numeric data to be factors. It's better to have them as plain-old characters.

This problem is often encountered when reading in data from an external data file. Let's import the file `common_names.csv`, which can be found at the GitHub repository for this book. This is data on the 100 most common names in America, and came from a story I wrote with my former colleague Mona Chalabi for FiveThirtyEight. The data has two columns, one for the name and the other (`perct_pop`) for it's share of the population.

Below we'll import the data using the "base R" function `read.csv` and then we'll attempt to calculate the number of characters in each name using the `nchar()` function. But it will fail. That's because the column `name` is not a charater data type. It's a factor, as we'll find out by using the helpful `str()` command, which tell us the structure of our dataframe.

```
common_names <- read.csv("common_names.csv")
# Attempt to calculate characters in each name
nchar(common_names$name)
```

```
## Error in nchar(common_names$name): 'nchar()' requires a character vector
```

```
str(common_names)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ name     : Factor w/ 100 levels "Aaron","Adam",..: 70 39 44 83 24 99 67 21 48 82 ...
##  $ perct_pop: num  0.01158 0.01022 0.00967 0.00949 0.00894 ...
```

This is frustrating and annoying. My advice is to have R, by default, interpret strings as charaters, not factors. There are several ways to accomplish this, a common one being to set the `stringsAsFactors` setting to `FALSE`. You can do this globally with the `options()` function, but I would **NOT** recommend that. Why? Becaue it can cause problems when sharing your code with others. You'd either be changing *their* global settings, which isn't polite; or your code could break when they run it.

Scratching that, you have two options. You can set `stringsAsFactors == FALSE` directly in your function call, if that function allows it (like `read.csv` does). Once we do this and run the `str()` function, notice how the data type of `name` is now a character, not a factor. See here:

```
common_names <- read.csv("common_names.csv", stringsAsFactors = FALSE)
str(common_names)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ name     : chr  "Michael" "James" "John" "Robert" ...
##  $ perct_pop: num  0.01158 0.01022 0.00967 0.00949 0.00894 ...
```

```
nchar(common_names$name)
```

```
##   [1]  7  5  4  6  5  7  4 11  6  7  6  6  7  8  7  7  8  5  4  9  6  6  6
##  [24]  5  5  7  7  5  5  7  4  7  4  4  5  5  7  6  6  4  5  8  8  6  8  8
##  [47]  5  6  6  6  7  9  5  7  4  6  7  5  6  5  4  7  7  5  7  5  3  8  7
##  [70]  6  5  6  5  9  8  6  8  7  8  5  6  5  6  6  5  5  7  6  4  9  7  5
```

```
##  [93]  9  6  6  9  5  6  4  4
```

But I'd argue there is an even better approach: use the `readr` package developed by Hadley Wickham. `readr` was installed with the `tidyverse` package we loaded earlier. It has lots of great functions for reading data into R. The analogous `readr` function to `read.csv` is `read_csv`. In addition to being faster, it interprets string data as characters (and not factors) **by default**. So no more typing `stringsAsFactors == FALSE`. Just use `readr`.

```
library(readr)
common_names <- read_csv("common_names.csv")
```

```
## Parsed with column specification:
## cols(
##   name = col_character(),
##   perct_pop = col_double()
## )
```

```
str(common_names)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    100 obs. of  2 variables:
##  $ name     : chr  "Michael" "James" "John" "Robert" ...
##  $ perct_pop: num  0.01158 0.01022 0.00967 0.00949 0.00894 ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 2
##   .. ..$ name     : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ perct_pop: list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

3.0.3 Subsetting

Another common source of R mistakes comes when subsetting your data. Say you want to select a column from a data frame. Well, you have a few options. You can use the `$` or `[` operators. But a common mistake is to use the `$` operator with a *variable* for the column yuo want to select. That is why you should use the `[` operator with variables. Let's try this out on the classic toy dataset `mtcars`, part of the `datasets` package installed with "base R."

```
mtcars$mpg # This works
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
column_to_select <- 'wt'
mtcars$column_to_select # This does NOT work
```

```
## NULL
```

```
mtcars['wt'] # This is how you use variables to subset
```

```
##                     wt
## Mazda RX4          2.620
## Mazda RX4 Wag      2.875
## Datsun 710         2.320
```

```
## Hornet 4 Drive      3.215
## Hornet Sportabout   3.440
## Valiant             3.460
## Duster 360          3.570
## Merc 240D           3.190
## Merc 230            3.150
## Merc 280            3.440
## Merc 280C           3.440
## Merc 450SE          4.070
## Merc 450SL          3.730
## Merc 450SLC         3.780
## Cadillac Fleetwood  5.250
## Lincoln Continental 5.424
## Chrysler Imperial   5.345
## Fiat 128            2.200
## Honda Civic         1.615
## Toyota Corolla      1.835
## Toyota Corona       2.465
## Dodge Challenger    3.520
## AMC Javelin         3.435
## Camaro Z28          3.840
## Pontiac Firebird    3.845
## Fiat X1-9           1.935
## Porsche 914-2       2.140
## Lotus Europa        1.513
## Ford Pantera L      3.170
## Ferrari Dino        2.770
## Maserati Bora       3.570
## Volvo 142E          2.780
```

Notice how subsetting `mtcars` with `$` vs. `[` has a different output; the latter includes rownames. That's because `$` returns the column as a vector, whereas the `[` returns the column of the dataframe as what it is – a list. If what you want is not a list, then you'll need to use the third type of subsetting operator, `[[`. Also, in addition to the `str()` function to examine the object, you can use `typeof()` too.

```
# Subsetting for a list
mtcars['wt']
```

```
##                      wt
## Mazda RX4           2.620
## Mazda RX4 Wag       2.875
## Datsun 710          2.320
## Hornet 4 Drive      3.215
## Hornet Sportabout   3.440
## Valiant             3.460
## Duster 360          3.570
## Merc 240D           3.190
## Merc 230            3.150
## Merc 280            3.440
## Merc 280C           3.440
## Merc 450SE          4.070
## Merc 450SL          3.730
## Merc 450SLC         3.780
## Cadillac Fleetwood  5.250
## Lincoln Continental 5.424
```

```
## Chrysler Imperial   5.345
## Fiat 128            2.200
## Honda Civic         1.615
## Toyota Corolla      1.835
## Toyota Corona       2.465
## Dodge Challenger    3.520
## AMC Javelin         3.435
## Camaro Z28          3.840
## Pontiac Firebird    3.845
## Fiat X1-9           1.935
## Porsche 914-2       2.140
## Lotus Europa        1.513
## Ford Pantera L      3.170
## Ferrari Dino        2.770
## Maserati Bora       3.570
## Volvo 142E          2.780
```

```
typeof(mtcars['wt'])
```

```
## [1] "list"
```

```
# Subsetting for a vector
mtcars[['wt']]
```

```
##  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
## [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520
## [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

```
typeof(mtcars[['wt']])
```

```
## [1] "double"
```

3.0.4 Missing values

R handles missing values with an `NA`, but it's often a source of frustration. Take the example below. Note: we're using the `c()` function – short for "combine" – to create a vector of numbers.

```
sample_data <- c(23, 7, 19, NA, -4, 12, 7)

# Test for NAs
is.na(sample_data)
```

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

```
sum(sample_data)
```

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

```
mean(sample_data)
```

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

As you see, some R functions return `NA` by default if one element is an `NA`. To fix this, one must either filter out the `NA` values or set the `na.rm` parameter to `TRUE`, which removes them.

```
# Filter out NAs
sum(sample_data[!is.na(sample_data)])
```

```
## [1] 64
```

```
# Tell function to remove NAs before calculating
sum(sample_data, na.rm = TRUE)
```

```
## [1] 64
```

One relevant warning about `NA` values: they don't appear by default in the useful `table()` function, which tallies the results of a single vector, factor or list. To have `table()` return `NA` values you must specify the `useNA` paramaeter to be `'ifany'` (in quotes).

```
table(sample_data) # Notice the NA is missing
```

```
## sample_data
## -4  7 12 19 23
##  1  2  1  1  1
```

```
table(sample_data, useNA = 'ifany')
```

```
## sample_data
##   -4    7   12   19   23 <NA>
##    1    2    1    1    1    1
```

Enough of these silly mistakes. Let's move on to how to write more elegant R code.

Chapter 4

Style

Let's not kid ourselves: reading code is often terrible. Especially when you wrote it ages ago and were too rushed (or lazy) to properly clean and document your work. That's why having good style is so important.

There are several good R style guides. I can recommend Google's R Style Guide. Hadley Wickham has great advice, too. Reading those guides is a great start. But what I want to focus on here is the importance of *readability* in writing R code, specifically **for data science**.

4.0.1 Embracing the power of piping with %>%

I'm of the strong opinion that the secret to readable R code comes down to one habit: liberal use of the `%>%` operator, or the "pipe" operator as its known. The `%>%` orginiated in the `maggrittr` package, but has become a de facto style of the `tidyverse` packages. The `%>%` allows you to write *chained* R commands. And it avoids ugly nested code.

Here's how to it works. We'll be using the data from the pro basketball (my favorite sport). FiveThirtyEight, my old employer, was fond of calculating Elo ratings for sports teams. Elo ratings originated in chess, and are a simple way of measuring quality. You can download the Elo ratings of *every* NBA team, before and after *every* NBA game, from the 1947 season through the 2015 season. The data is online here (and can also be found on this book's GitHub repository).

First, let's make sure the `tidyverse` package is loaded. Next, we'll read the data into R using the `read_csv` function from the `readr` package. This might take your computer some time – it's over 126,000 games! Then we'll calculate the average post-game Elo rating for all teams in the dataset (the pre-game elo rating is `elo_i`, the post-game rating is `elo_n`). I'll show you two ways to do this: one straightforward way using `mean()` and anothering using the pipe (`%>%`) operator.

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------------------------------------
```

```
## filter(): dplyr, stats
## lag():    dplyr, stats
```

```
# Load data
nba_elo_data <- read_csv("https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallel
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   gameorder = col_integer(),
##   `_iscopy` = col_integer(),
##   year_id = col_integer(),
##   seasongame = col_integer(),
##   is_playoffs = col_integer(),
##   pts = col_integer(),
##   elo_i = col_double(),
##   elo_n = col_double(),
##   win_equiv = col_double(),
##   opp_pts = col_integer(),
##   opp_elo_i = col_double(),
##   opp_elo_n = col_double(),
##   forecast = col_double()
## )
```

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

```
# Calculate average Elo rating over time
mean(nba_elo_data$elo_n)
```

```
## [1] 1495.236
```

```
# Do the same, using pipes
nba_elo_data %>% summarize(avg_elo = mean(elo_n))
```

```
## # A tibble: 1 × 1
##    avg_elo
##      <dbl>
## 1 1495.236
```

Wait, what? Piping requires writing more code?!

Hang on a second, this example is just meant to illustrate how the `%>%` operator works. Here the pipe is passing the `nba_elo_data` dataframe to the `summarize()` function (from the `dplyr` package). Not having to specify the dataframe you're operating on for each function will come in handy when you're calling multiple functions.

Let's continue to work with the `nba_elo_data` dataset, and continue using basic `dplyr` functions but writing code with and without pipes. First, let's calculate the highest Elo rating for each team – or franchise, really – by grouping each game according to the `fran_id` column. Notice that with pipes, you can place each piece of code on a new line, as long as the line ends with the `%>%` operator.

```
library(tidyverse)

# Code without pipes
summarize(group_by(nba_elo_data, fran_id), best_elo = max(elo_n))
```

```
## # A tibble: 53 × 2
##      fran_id best_elo
##        <chr>    <dbl>
## 1  Baltimore 1513.935
## 2    Bombers 1467.990
```

```
## 3      Bucks 1757.149
## 4      Bulls 1853.104
## 5   Capitols 1563.444
## 6  Cavaliers 1765.244
## 7    Celtics 1815.692
## 8   Clippers 1743.113
## 9   Colonels 1672.633
## 10   Condors 1499.175
## # ... with 43 more rows
```

```
# Code with pipes
nba_elo_data %>%
  group_by(fran_id) %>%
  summarize(best_elo = max(elo_n))
```

```
## # A tibble: 53 × 2
##      fran_id best_elo
##        <chr>    <dbl>
## 1  Baltimore 1513.935
## 2    Bombers 1467.990
## 3      Bucks 1757.149
## 4      Bulls 1853.104
## 5   Capitols 1563.444
## 6  Cavaliers 1765.244
## 7    Celtics 1815.692
## 8   Clippers 1743.113
## 9   Colonels 1672.633
## 10   Condors 1499.175
## # ... with 43 more rows
```

Ahhh, see? The code with pipes is far easier on the eye. We can take this point a step further: let's filter out all the games before 1980 and then sort the resulting dataframe from highest to lowest peak franchine Elo rating. Notice how ugly the nested code becomes when the pipes aren't used.

```
library(tidyverse)

# Code without pipes
arrange(summarize(group_by(filter(nba_elo_data, year_id >= 1980), fran_id), best_elo = max(elo_n)), des
```

```
## # A tibble: 30 × 2
##      fran_id best_elo
##        <chr>    <dbl>
## 1      Bulls 1853.104
## 2   Warriors 1822.288
## 3    Celtics 1815.692
## 4     Lakers 1789.993
## 5    Pistons 1788.091
## 6      Magic 1781.763
## 7     Sixers 1777.069
## 8       Heat 1774.346
## 9  Mavericks 1773.131
## 10     Spurs 1771.188
## # ... with 20 more rows
```

```
# Code with pipes
nba_elo_data %>%
```

```
  filter(year_id >= 1980) %>%
  group_by(fran_id) %>%
  summarize(best_elo = max(elo_n)) %>%
  arrange(desc(best_elo))
```

```
## # A tibble: 30 × 2
##      fran_id best_elo
##        <chr>    <dbl>
## 1      Bulls 1853.104
## 2   Warriors 1822.288
## 3    Celtics 1815.692
## 4     Lakers 1789.993
## 5    Pistons 1788.091
## 6      Magic 1781.763
## 7     Sixers 1777.069
## 8       Heat 1774.346
## 9  Mavericks 1773.131
## 10     Spurs 1771.188
## # ... with 20 more rows
```

The fist bit of code is so messy and hard to understand that I made a mistake coding this example!

Piping is the key to clear, readable R code for data science.

4.0.2 The right match

Another common R stylistic pitfall is misuing the `match()` function, either by not using the more readable `%in%` operator or by using it to join data between different dataframes.

The `match()` function is very useful. It takes as its input two vectors and returns the indices where matches of the elements in the first vector (if any) are found in the elements of the second vector.

```
large_cities <- c("New York", "Los Angeles", "Chicago", "Houston", "Philadelphia", "Phoenix")

warm_cities <- c("Miami", "New Orleans", "Houston", "Phoenix", "San Diego", "Los Angeles")

match(large_cities, warm_cities)
```

```
## [1] NA  6 NA  3 NA  4
```

```
match(warm_cities, large_cities)
```

```
## [1] NA NA  4  6 NA  2
```

In many cases, though, you'll want to use the `%in%` operator for matching, rather than the `match()` function itself, because it's aesthetically nicer. But the `%in%` operator works a little differently, returning a *logical* vector, with `TRUE`/`FALSE` as to whether there is a match at all.

```
large_cities <- c("New York", "Los Angeles", "Chicago", "Houston", "Philadelphia", "Phoenix")

warm_cities <- c("Miami", "New Orleans", "Houston", "Phoenix", "San Diego", "Los Angeles")

large_cities %in% warm_cities
```

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

Using the `match()` function to join data to a dataframe is a mistake I often made in my early days of R coding. Taking two dataframes – `birthdays` and `cities` – and joining the later data by the `name` column. Here is

the ugly way to do: using `match()` to subset out the `city` column in the correct order for the `birthdays` dataframe:

```
birthdays <- data.frame(
  name = c('Steve', 'Laura', 'Kim'),
  birthday = c('2/17/71', '10/4/83', '6/28/66')
)

cities <- data.frame(
  name = c('Laura', 'Kim', 'Steve'),
  city = c('Chicago', 'Houston', 'Seattle')
)

# Find match between two dataframes according to name column
match(birthdays$name, cities$name)
```

```
## [1] 3 1 2
```

```
# Use that match to join city data to birthdays dataframe
birthdays$city <- cities[match(birthdays$name, cities$name),]$city

birthdays
```

```
##    name birthday    city
## 1 Steve  2/17/71 Seattle
## 2 Laura  10/4/83 Chicago
## 3   Kim  6/28/66 Houston
```

Ugh. That was terrible. Don't do this. Why? You can easily flip the order of the two vectors in your match function and it will it return the data sorted incrrectly. But, more importantly, there is a stylically superior way to join data – using the join-functions from the `dplyr` package.

Here I'll use the `left_join()` function to extract join the `city` column from `cities` to the `birthdays` dataframe. All you must do is specify the `by =` parameter (which is the column or columns to join by).

```
library(dplyr)

birthdays <- data.frame(
  name = c('Steve', 'Laura', 'Kim'),
  birthday = c('2/17/71', '10/4/83', '6/28/66')
)

cities <- data.frame(
  name = c('Laura', 'Kim', 'Steve'),
  city = c('Chicago', 'Houston', 'Seattle')
)

birthdays %>%
  left_join(cities, by = 'name')
```

```
##    name birthday    city
## 1 Steve  2/17/71 Seattle
## 2 Laura  10/4/83 Chicago
## 3   Kim  6/28/66 Houston
```

There are other join functions in `dplyr` – `right_join()`, `inner_join()`, `outer_join()` and so on. Use those to join data across separate dataframes. Don't use `match()`. The benefits of using the join-functions are several. There is less risk of a "silent" error, for one. But it's also far easier to interpret.

Chapter 5

Stats

R is built to do stats. For my money, it's the best data analysis tool there is. But users commonly make mistakes doing statistics in R. Here are some and how to avoid them.

5.0.1 Don't regress when regressing

Regression analysis is a staple method in data science. In R, you'll often use the workhorse `lm()` function for doing linear regression and `glm()` for logistic regression and more advanced techniques.

Let's load the `nba_elo_data.csv` data again and run a regressions on a team's Elo rating at the start of a game (`elo_i`) against its point margin (which we'll create by subtracting `pts` from `opp_pts`, using the `mutate()` function).

```
library(tidyverse)

# Load data
nba_elo_data <- read_csv("https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallel
```

```
# Add new column, pts_margin, using the mutate() function
new_elo_data <- nba_elo_data %>%
  mutate(pts_margin = pts - opp_pts)

lm(data = new_elo_data, formula = pts_margin ~ elo_i)
```

```
##
## Call:
## lm(formula = pts_margin ~ elo_i, data = new_elo_data)
##
## Coefficients:
## (Intercept)        elo_i
##   -46.91730      0.03138
```

Not surprising: an NBA team's pre-game Elo rating is *positively* associated with its margin of victory.

But the output, `elo_regression`, is hard to read, And it masks more detailed output from the regression. We'll assign the `lm()` output (which is a model object) to the variable named `elo_regression`. Using the customary `summary()` function on the model object reveals more details – but it's still hard to decipher.

```
elo_regression <- lm(data = new_elo_data, formula = pts_margin ~ elo_i)
```

```
# Printing the raw model output
elo_regression
```

```
##
## Call:
## lm(formula = pts_margin ~ elo_i, data = new_elo_data)
##
## Coefficients:
## (Intercept)        elo_i
##   -46.91730      0.03138
```

```
# Printing the detailed model output
summary(elo_regression)
```

```
##
## Call:
## lm(formula = pts_margin ~ elo_i, data = new_elo_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -67.818  -8.544   0.032   8.523  67.526
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.692e+01  4.855e-01  -96.65   <2e-16 ***
## elo_i        3.138e-02  3.238e-04   96.92   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.9 on 126312 degrees of freedom
## Multiple R-squared:  0.06922,    Adjusted R-squared:  0.06921
## F-statistic:  9393 on 1 and 126312 DF,  p-value: < 2.2e-16
```

Now let's actually use the regression output. A common beginner mistake is to make predictions from a regression by *directly* accessing the model coefficients and then *manually* calculating the estimator (or "y-hat").

```
# Print regression coeficients
coef(elo_regression)
```

```
##  (Intercept)        elo_i
## -46.91730071   0.03137786
```

```
# BAD -- manually making a prediction
example_team_elo <- 1500
coef(elo_regression)[1] + (coef(elo_regression)[2] * example_team_elo)
```

```
## (Intercept)
##   0.1494824
```

This is a big no-no. It's far more simple – and less error-prone – to use the `predict()` function instead. Under this workflow, you'd provide `predict()` with two necessary inputs: (1) the model object you've generated from `lm()` and (2) a dataframe of new data to make predictions on (with the paramaeter name `newdata`). Here's a better way to make a prediction on the point margin of an NBA team with a 1500 Elo rating.

```
# Create data frame to use for predictions
new_data <- data.frame(elo_i = 1500)
```

```
# GOOD -- using the predict() function
predict(elo_regression, newdata = new_data)
```

```
##         1
## 0.1494824
```

5.0.2 Use `broom` to tidy your model outputs

There is an even better workflow for working with the output of R regression models, especially when you're creating a lot of them at once. The wonderful package `broom`, by David Robinson, is here to save you. What `broom` does is simple but powerful: It stores model output data in a standardized data frame.

What if you want to extract the p-value from your regression? When using `summary()` on a model object, what's returned is OK enough to read, but a terrible format *to use*. The `broom` package has a useful function, `tidy()`, which nicely organizes the terms of your regression, their estiamtes and standard errors, test statistics and p-values – all in one place.

```
library(broom)

tidy(elo_regression)
```

```
##          term     estimate    std.error statistic p.value
## 1 (Intercept) -46.91730071 0.4854565302 -96.64573       0
## 2       elo_i   0.03137786 0.0003237596  96.91715       0
```

This is especially useful if you're testing a lot of regression models. Let's group each NBA franchise (using `fran_id`) and run a regression the same regression as before.

To do this for each franchise, we'll use the helpful `do()` function. First, though, we'll run the regression using `lm()` and store the regression outputs in a dataframe using `tidy()`. What results is a clean dataframe for all 53 franchises in NBA history. One clarifying note: the `.` used to specify the dataset for `lm()` to use refers to the dataframe being piped (`%>%`) to it, or `new_elo_data`.

```
new_elo_data %>%
  group_by(fran_id) %>%
  do(tidy(lm(formula = pts_margin ~ elo_i, data = .)))
```

```
## Source: local data frame [106 x 6]
## Groups: fran_id [53]
##
##      fran_id        term     estimate    std.error   statistic
##        <chr>       <chr>        <dbl>        <dbl>       <dbl>
## 1  Baltimore (Intercept) -63.58332224 13.161953246  -4.8308424
## 2  Baltimore       elo_i   0.04307166  0.009449932   4.5578808
## 3    Bombers (Intercept)  59.43686404 26.951049832   2.2053636
## 4    Bombers       elo_i  -0.04325854  0.019260027  -2.2460271
## 5      Bucks (Intercept) -53.85436245  2.926199735 -18.4041991
## 6      Bucks       elo_i   0.03618180  0.001929639  18.7505475
## 7      Bulls (Intercept) -51.98972815  2.306071267 -22.5447188
## 8      Bulls       elo_i   0.03482617  0.001515665  22.9774815
## 9   Capitols (Intercept)  21.71334358 25.122186049   0.8643095
## 10  Capitols       elo_i  -0.01347413  0.017132549  -0.7864641
## # ... with 96 more rows, and 1 more variables: p.value <dbl>
```

5.0.3 Logitstic troubles

Logistic regressions are another source of R frustration. A beginner mistake is to think you can just use `lm()`. You can't.

Let's try to run a logistic regression on how an NBA team's pre-game Elo rating predicts the likehlihood they'll win the game (`game_result`). It's going to fail, though, because we're using `lm()`.

```
lm(data=new_elo_data, formula = game_result ~ elo_i)
```

```
## Warning in model.response(mf, "numeric"): NAs introduced by coercion
```

```
## Error in lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...): NA/NaN/Inf in 'y'
```

Maybe R's most powerful function is `glm()`, which stands for generalized linear model. And it's this you'll need for a logistic regression. Also requried is that you specifcy the `family` parameter within the `glm()` function. A common setting for logistic regression is "binmoial," as logistic is measuring outcomes like TRUE/FALSE, Democratc/Republican, Win/Loss.

But your data must be properly formatted first. Notice how this otherwise correct specfication fails. (Hint: it's because `game_result` is currently a character vector with "W" or "L".)

```
glm(data=new_elo_data, formula = game_result ~ elo_i, family = "binomial")
```

```
## Error in eval(expr, envir, enclos): y values must be 0 <= y <= 1
```

Let's change that to be 1s and 0s, and then re-run the regression.

```
library(tidyverse)

elo_data_logit <- new_elo_data %>%
  mutate(game_result_logit = ifelse(game_result == "W", 1, 0))

glm(data=elo_data_logit, formula = game_result_logit ~ elo_i, family = "binomial")
```

```
##
## Call:  glm(formula = game_result_logit ~ elo_i, family = "binomial",
##     data = elo_data_logit)
##
## Coefficients:
## (Intercept)        elo_i
##   -6.547575     0.004378
##
## Degrees of Freedom: 126313 Total (i.e. Null);  126312 Residual
## Null Deviance:       175100
## Residual Deviance: 168000    AIC: 168100
```

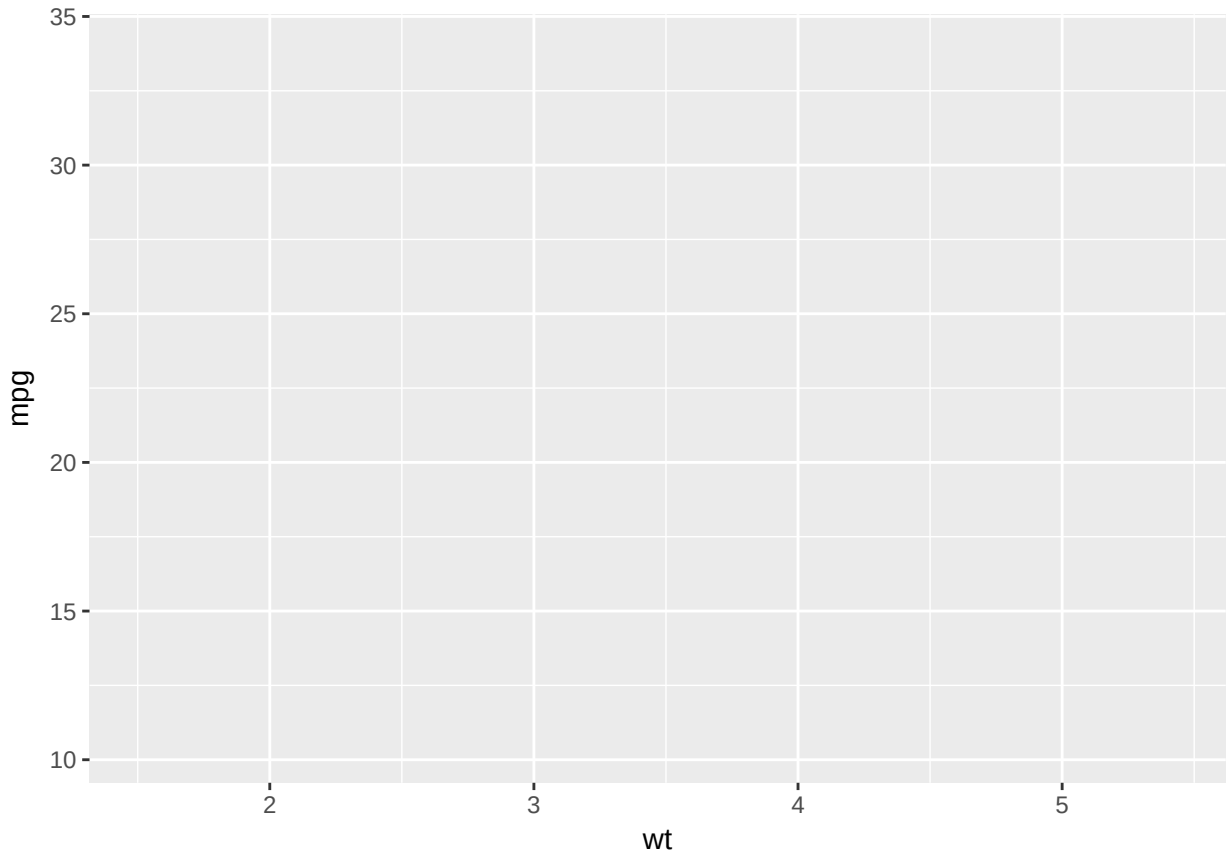
Hooray! It worked.

5.0.4 ggplot2 screwups

Hadley Wickham's `ggplot2` is a gem. The pacakge allows R coders to create elegant graphics through an approach that utilizes a "grammar of graphics." I won't cover all the intricacies of `ggplot2` here, but I'll touch on some common mistakes I've made.

Perhaps the most common `ggplot2` mistake is forgetting where to place the `+` symbol that "chains" together ggplot commands (often called "layers"). (Hadley Wickham cites this as a common mistake in his fantastic book R for Data Science.) The error occurs because the `+` symbol should be placed at the *end* of the line of code.

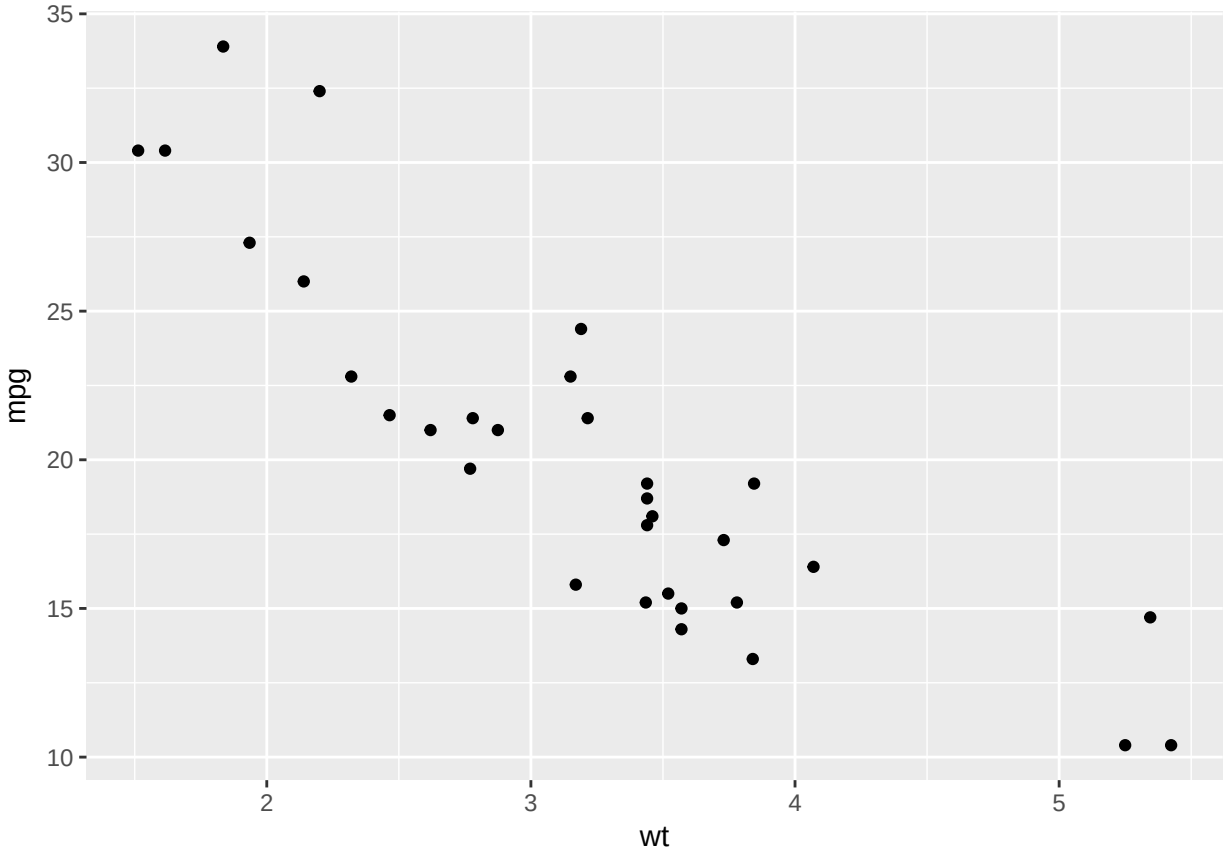
```
library(ggplot2)

# Incorrect
ggplot(data = mtcars, aes(x = wt, y = mpg))
```




```
  + geom_point()
```

```
## Error in +geom_point(): invalid argument to unary operator
```

```
# Correct
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```



Beyond that silly mistake, I've made plenty of other ones in `ggplot2`. A common one involves the `geom_bar()` layer for making bar plots.

Here we're going to create a dataframe of each NBA franchise's peak Elo rating (similar to what we did in chapter 3). We'll then use this dataframe, `fran_max_elo`, to make a simple bar chart. (For clarity, we'll flip the x and y coordinates of the chart with the `coord_flip()` function.) All this will be chained together with the `+` symbol and, finally, plotted with `geom_bar()`. But, by default, it won't work.

```
fran_max_elo <- nba_elo_data %>%
  filter(year_id >= 1980) %>%
  group_by(fran_id) %>%
  summarize(best_elo = max(elo_n)) %>%
  arrange(desc(best_elo))

ggplot(data = fran_max_elo, aes(x = fran_id, y = best_elo)) +
  coord_flip() +
  geom_bar()
```

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

The missing ingredient here is changing the `stat` paramaeter of the `geom_bar()` layer to be set to "identity". This insures that the bars are not counting, or aggregating, the variable supplied for y – but instead are interpreting it literally.

```
ggplot(data = fran_max_elo, aes(x = fran_id, y=best_elo)) +
  coord_flip() +
  geom_bar(stat="identity")
```

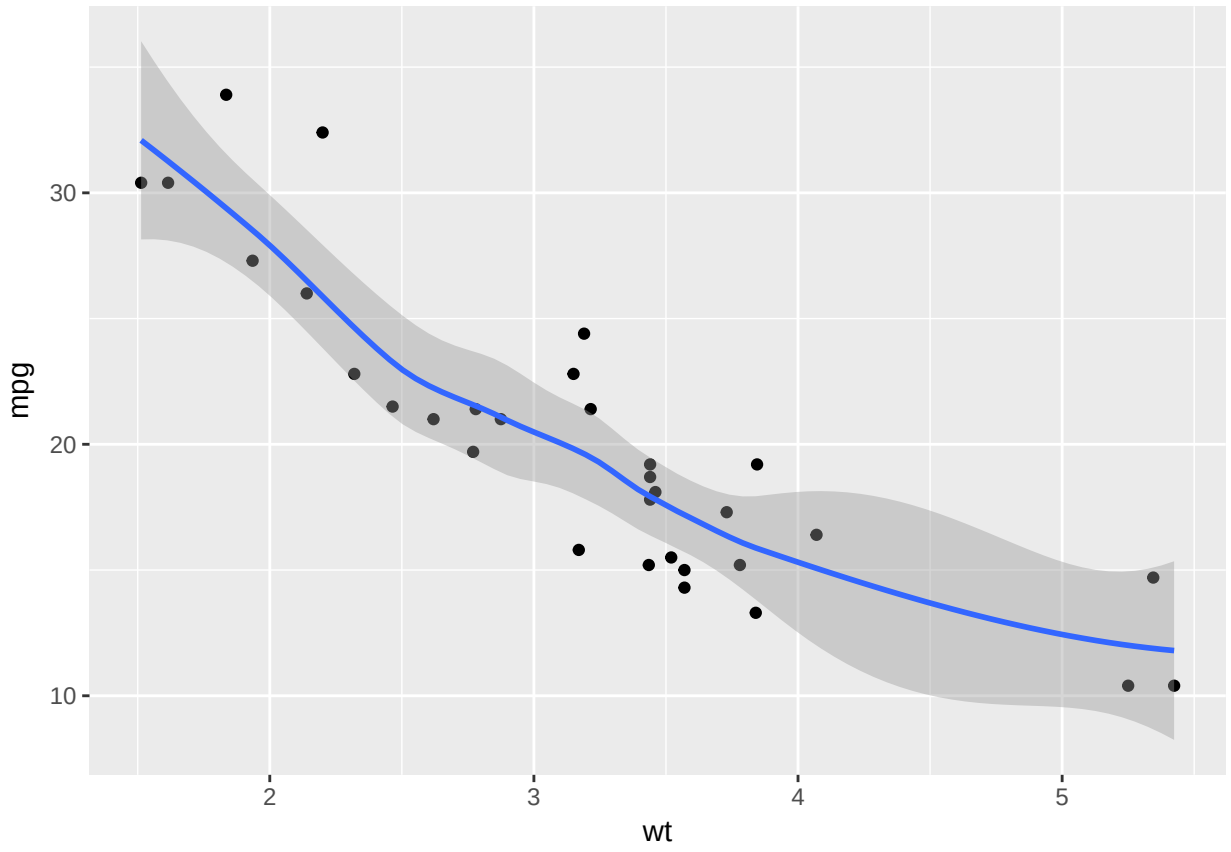
```
## Error in eval(expr, envir, enclos): object 'fran_id' not found
```

Another common mistake is to trust `ggplot2` default settings when using the `geom_smooth()` or `stat_smooth()` functions. Without any additional specification, it will layer on your graphic a Loess curve. To instead get a linear line of best fit, you must specify with `method="lm"`.

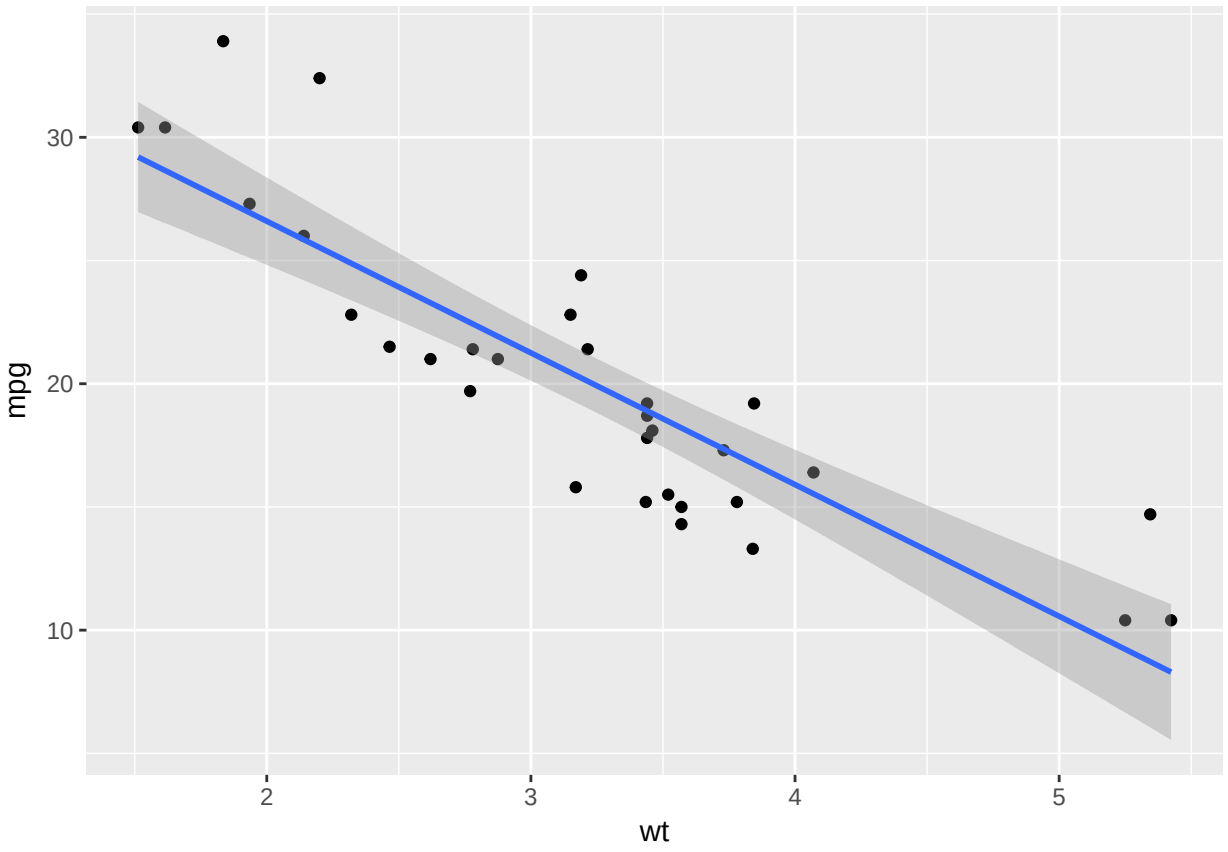
Also, notice in the code below I stop specifying the `data` parameter in the `ggplot` base layer. Instead, per my style suggestions in chapter 3, I use the `%>%` operator. You'll also pick up the interchanging symbols, placed at the end of the code line.

```
library(tidyverse)

# Loess curve by default
mtcars %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess'
```




```
# Linear smoothing
mtcars %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point() +
  geom_smooth(method="lm")
```


mpg
30
20
10
2
3
4
5
wt

Chapter 6

Surprises

Software developers don't like to be surprised. So no wonder many are frustrated with R. It behaves in ways inconsistent with more traditional programming languages. This behavior can sometimes be chalked up to the idiosyncratic designs within R. But, frankly, some of its behavior is inconsistent or outright confusing.

6.0.1 Function masking

"There are only two hard things in Computer Science: cache invalidation and naming things. – Phil Karlton

Yes, that's a programming joke. (Hilarious, right? OK, I'll stop.) The joke has a lot of truth to it: naming things is hard.

In R, you'll probably run into problems with your namespace. In particular, you'll load two identially-named functions from different pacakges. And it will call all sorts of problems.

A classic instance of this comes from using multiple generations of Hadley Wickham's data manipulation packages. Using both `plyr` and `dplyr`, in particular, illustrates the problem of "function masking" – overwriting the use of one function with another one with the same name. Pay attention to the warning messages.

```
library(dplyr)
library(plyr)
```

Now see what happens when use the `mtcars` dataset and perform a simple calculation using the `summarize()` function.

```
library(dplyr)
library(plyr)
# summarize() is "masked""
mtcars %>%
  group_by(cyl) %>%
  summarize(avg_mpg = mean(mpg))
```

```
##    avg_mpg
## 1 20.09062
```

Now that is **not** what we wanted! It was supposed to return the `avg_mpg` for cars grouped by the number of cylinders (`cyl`).

To specify which `summarize()` function you want to use exactly, preface your function with package name and two colons (`:`), like this:

```
# summarize() from dplyr
mtcars %>%
  group_by(cyl) %>%
  dplyr::summarize(avg_mpg = mean(mpg))
```

```
## # A tibble: 3 × 2
##     cyl  avg_mpg
##   <dbl>    <dbl>
## 1     4 26.66364
## 2     6 19.74286
## 3     8 15.10000
```

6.0.2 Annoying levels

Note: This "mistake" suggestion comes from Jose Manuel Vera.

Factors are an R data type that often causes problems (as we covered in chapter 2). As you'll see in the example below, we'll create a dummy dataframe that has a `letters` column that are factors.

```
df <- data.frame(letters=letters[1:5],numbers=seq(1:5))

df
```

```
##   letters numbers
## 1       a       1
## 2       b       2
## 3       c       3
## 4       d       4
## 5       e       5
```

```
str(df)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ letters: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
##  $ numbers: int  1 2 3 4 5
```

With factors, you can discover the pre-specified categories with the `levels()` function. But it's a mistake to assume these levels disappear once said dataframe is subsetted.

```
levels(df$letters)
```

```
## [1] "a" "b" "c" "d" "e"
```

```
subdf <- subset(df, numbers <=3)
subdf
```

```
##   letters numbers
## 1       a       1
## 2       b       2
## 3       c       3
```

```
# check levels
levels(subdf$letters)
```

```
## [1] "a" "b" "c" "d" "e"
```

Weird – you'd think that the levels "d" and "e" would disappeard after subetting out those vaules. But they don't. To permanently remove unneeded levels use the `droplevels()` function.

```
droplevels(subdf$letters)
```

```
## [1] a b c
## Levels: a b c
```

6.0.3 Escape characters

Like any programming language, R supports regular expressions – a special syntax and set of methods for searching strings. You may use the `regexpr()` function, or one of its siblings like `gsub()` or `grep()`. But in using these functions, you'll have to use an escape character – a character that tells the function to interpret the *subsequent* character differently. In R, as in most languages, the escape character is the backslash (\).

Take this examle: we're trying to remove the `$` characters from a vector of prices. (So that the resulting data is just numeric data.) We'll do this using the `gsub()` function – removing every instance of `$`, or replace it with the empty character `""`. Type `?gsub` to learn more about this function.

```
prices <- c("$217", "$840", "$503")

# Incorrect: No escape character \
gsub("$", "", prices)
```

```
## [1] "$217" "$840" "$503"
```

Ah, well that did precisely nothing. Let's add the escape character before the `$`.

```
# Incorrect: Only one escape character \
gsub("\$", "", prices)
```

```
## Error: '\$' is an unrecognized escape in character string starting ""\$"
```

Scratching your head? That's because in R the backslash *itself* needs to be escaped.

So often the correct number of escape characters is two `\\`.

```
# Correct: two escape characters \\
gsub("\\$", "", prices)
```

```
## [1] "217" "840" "503"
```

6.0.4 Lists and double-bracket notation `[[]]`

Of R's data types, a list is the most flexible. It can take hetergenous input (a numeric, a charactacter and a boolean all at once). But accessing values within a list often comes as a surprise for beginners.

```
customers <- list(
  names = c("Jen", "Mark", "Mindy"),
  age = c(41, 17, 33),
  female = c(TRUE, FALSE, TRUE)
  )

# Extract ages
customers[2]
```

```
## $age
## [1] 41 17 33
```

```
# But it's a list!?
str(customers[2])
```

```
## List of 1
##  $ age: num [1:3] 41 17 33
```

When subsetting a list using single-bracket notation (`[ ]`), another list is returned. To get the actual atomic data type held with that list you need to use double-bracket notation (`[[ ]]`).

```
# Extract ages as a numeric vector
customers[[2]]
```

```
## [1] 41 17 33
```

```
str(customers[[2]])
```

```
##  num [1:3] 41 17 33
```