STAT 545

Data wrangling, exploration, and analysis with R

Jenny Bryan The STAT 545 TAs

Contents

Welcome to STAT 545 **9**

History and future 9

Other contributors 10

Colophon 10

License 13

I Get your R act together 15

1 Install R and RStudio **17**

1.1 R and RStudio 17

1.2 Testing testing 18

1.3 Add-on packages 18

1.4 Further resources 19

2 R basics and workflows **21**

2.1 Basics of working with R at the command line and RStudio goodies 21

2.2 Workspace and working directory 24

2.3 RStudio projects 26

2.4 Stuff 29

II Version control and R Markdown 31

Overview **33**

3 Git, GitHub, and RStudio **35**

4 R Markdown **37**

III Data analysis 1 **39**

5 Basic care and feeding of data in R **41**

5.1 Buckle your seatbelt . . . 41
5.2 Data frames are awesome . . . 42
5.3 Get the Gapminder data . . . 42
5.4 Meet the `gapminder` data frame or "tibble" . . . 42
5.5 Look at the variables inside a data frame . . . 48
5.6 Recap . . . 51

6 Introduction to dplyr **53**

6.1 Intro . . . 53
6.2 Think before you create excerpts of your data … . . . 55
6.3 Use `filter()` to subset data row-wise . . . 56
6.4 Meet the new pipe operator . . . 57
6.5 Use `select()` to subset the data on variables or columns. . . . 58
6.6 Revel in the convenience . . . 59
6.7 Pure, predictable, pipeable . . . 60
6.8 Resources . . . 61

7 Single table dplyr functions **63**

7.1 Where were we? . . . 63
7.2 Load dplyr and gapminder . . . 63
7.3 Create a copy of `gapminder` . . . 64
7.4 Use `mutate()` to add new variables . . . 65
7.5 Use `arrange()` to row-order data in a principled way . . . 67
7.6 Use `rename()` to rename variables . . . 69
7.7 `select()` can rename and reposition variables . . . 69

7.8 group_by() is a mighty weapon 70
7.9 Grouped mutate 74
7.10 Grand Finale 78
7.11 Resources 79

8 Tidy data 81

9 Writing and reading files 83

9.1 File I/O overview 83
9.2 Load the tidyverse 84
9.3 Locate the Gapminder data 85
9.4 Bring rectangular data in 85
9.5 Compute something worthy of export 86
9.6 Write rectangular data out 87
9.7 Invertibility 88
9.8 Reordering the levels of the country factor 89
9.9 saveRDS() and readRDS() 89
9.10 Retaining factor levels upon re-import 90
9.11 dput() and dget() 91
9.12 Other types of objects to use dput() or saveRDS() on 93
9.13 Clean up 93
9.14 Pitfalls of delimited files 93
9.15 Resources 94

IV Data analysis 2 95

10 Be the boss of your factors 97

10.1 Factors: where they fit in 97
10.2 The forcats package 98
10.3 Load forcats and gapminder 98
10.4 Factor inspection 98
10.5 Dropping unused levels 99

10.6 Change order of the levels, principled 100
10.7 Change order of the levels, “because I said so” 103
10.8 Recode the levels 103
10.9 Grow a factor 104

11 Character vectors 107

11.1 Character vectors: where they fit in 107
11.2 Resources 107
11.3 Load the tidyverse, which includes stringr 109
11.4 Regex-free string manipulation with stringr and tidyr 109
11.5 Regular expressions with stringr 115

12 Character encoding 123

12.1 Resources 123
12.2 Translating two blog posts from Ruby to R 123
12.3 What is an encoding? 124
12.4 A three-step process for fixing encoding bugs 126
12.5 How to Get From Theyâ€™re to They’re 127

13 Dates and times 131

13.1 Date-time vectors: where they fit in 131
13.2 Resources 131
13.3 Load the tidyverse and lubridate 132
13.4 Get your hands on some dates or date-times 132
13.5 Get date or date-time from character 133
13.6 Build date or date-time from parts 133
13.7 Get parts from date or date-time 133
13.8 Arithmetic with date or date-time 133
13.9 Get character from date or date-time 134

V Data analysis 3 **135**

14 When one tibble is not enough **137**
14.1 Typology of data combination tasks . . . 138
14.2 Bind . . . 139
14.3 Joins in dplyr . . . 143
14.4 Table Lookup . . . 143

15 Join two tables **145**
15.1 Why the cheatsheet . . . 145
15.2 The data . . . 145
15.3 `inner_join(superheroes, publishers)` . . . 146
15.4 `semi_join(superheroes, publishers)` . . . 149
15.5 `left_join(superheroes, publishers)` . . . 152
15.6 `anti_join(superheroes, publishers)` . . . 155
15.7 `inner_join(publishers, superheroes)` . . . 157
15.8 `semi_join(publishers, superheroes)` . . . 160
15.9 `left_join(publishers, superheroes)` . . . 162
15.10 `anti_join(publishers, superheroes)` . . . 166
15.11 `full_join(superheroes, publishers)` . . . 168

16 Table lookup **173**
16.1 Load gapminder and the tidyverse . . . 173
16.2 Create mini Gapminder . . . 173
16.3 Dorky national food example. . . . 174
16.4 Lookup national food . . . 174
16.5 World's laziest table lookup . . . 176

VI R as a programming language **179**

17 R objects and indexing **181**
17.1 Vectors are everywhere . . . 181

17.2 Indexing a vector 184
17.3 Lists hold just about anything 185
17.4 Creating a data.frame explicitly 189
17.5 Indexing arrays, e.g. matrices 191
17.6 Creating arrays, e.g. matrices 194
17.7 Putting it all together…implications for data.frames 197
17.8 Table of atomic R object flavors 199

18 Write your own R functions, part 1 201

18.1 What and why? 201
18.2 Load the Gapminder data 201
18.3 Max - min 202
18.4 Get something that works 202
18.5 Turn the working interactive code into a function 203
18.6 Test your function 204
18.7 Check the validity of arguments 205
18.8 Wrap-up and what's next? 207
18.9 Resources 207

19 Write your own R functions, part 2 209

19.1 Where were we? Where are we going? 209
19.2 Load the Gapminder data 209
19.3 Restore our max minus min function 209
19.4 Generalize our function to other quantiles 210
19.5 Get something that works, again 210
19.6 Turn the working interactive code into a function, again 211
19.7 Argument names: freedom and conventions 211
19.8 What a function returns 212
19.9 Default values: freedom to NOT specify the arguments 213
19.10 Check the validity of arguments, again 213
19.11 Wrap-up and what's next? 214
19.12 Resources 214

Welcome to STAT 545

Learn how to:

- explore, groom, visualize, and analyze data,
- make all of that reproducible, reusable, and shareable,
- using R.

This site is about everything that comes up during data analysis **except for statistical modelling and inference**. This might strike you as strange, given R's statistical roots. First, let me assure you we believe that modelling and inference are important. But the world already offers a lot of great resources for doing statistics with R.

The design of STAT 545 was motivated by the need to provide more balance in applied statistical training. Data analysts spend a considerable amount of time on project organization, data cleaning and preparation, and communication. These activities can have a profound effect on the quality and credibility of an analysis. Yet these skills are rarely taught, despite how important and necessary they are. STAT 545 aims to address this gap.

History and future

These materials originated in the STAT 545 course at the University of British Columbia:

> "The STAT 545 course became notable as an early example of a data science course taught in a statistics program. It is also notable for its focus on teaching using modern R packages, Git and GitHub, its extensive sharing of teaching materials openly online, and its strong emphasis on practical data cleaning, exploration, and visualization skills, rather than algorithms and theory."
>
> — Wikipedia

The main author, Jenny Bryan (jennybryan.org), developed this version of STAT 545 as a professor at UBC. She has since joined RStudio as a Software Engineer, on the tidyverse and r-lib teams and is an adjunct professor at UBC. In September 2019, we (amicably) created separate spaces for the ongoing maintenance of this content and the continued offerings of STAT 545 at UBC (https://stat545.stat.ubc.ca), which is alive and well.

We plan to continue maintaining these resources, as they are still used in STAT 545 at UBC and by people teaching themselves R. Some topics have since been developed more fully elsewhere and we may link out to those resources. For example, the Git and GitHub content of STAT 545 eventually grew into its own website: happygitwithr.com. Some material has been retired, but is archived in the repository of the old website. Finally, the new website has URLs that are more human-friendly; we believe we created the necessary redirects, so we don't break other people's links. If you think we've missed one, please let us know in an issue.

Other contributors

Several STAT 545 TAs were instrumental in the development of these materials and members of the RStudio Education Team ported the original website into the modern and more maintainable framework we enjoy today:

- TAs who contributed content: Dean Attali (web applications with Shiny), Julia Gustavsen (Shiny), Shaun Jackman (automating workflows), Bernhard Konrad (system setup, package development, the shell), Gloria Li (regular expressions), Andrew MacDonald (getting data from the web), Kieran Samuk (regular expressions)
- RStudio: Alison Hill (https://alison.rbind.io) and intern Grace Lawley (https://grace.rbind.io) lead the heroic effort to port a vintage R Markdown website into bookdown. Intern Desirée De Leon (https://desiree.rbind.io) contributed design expertise.

Colophon

This book was written in bookdown inside RStudio. The website stat545.com is hosted with Netlify, and automatically updated after every commit by Travis-CI. The complete source is available from GitHub.

The STAT 545 logo and the book style was designed by Desirée De Leon.

This version of the book was built with:

```
#> Finding R package dependencies ... Done!
```

```
#>  setting  value
#>  version  R version 4.0.2 (2020-06-22)
#>  os       macOS Catalina 10.15.7
#>  system   x86_64, darwin17.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       Europe/Berlin
#>  date     2020-10-27
```

Along with these packages:

Search:

package	loadedversion	date	source
All	All	All	All
askpass		2019-01-13	CRAN (R 4.0.0)
assertthat	0.2.1	2019-03-21	CRAN (R 4.0.0)
backports	1.1.10	2020-09-15	CRAN (R 4.0.2)
base64enc		2015-07-28	CRAN (R 4.0.0)
BH		2020-01-08	CRAN (R 4.0.0)
blob	1.2.1	2020-01-20	CRAN (R 4.0.0)
bookdown	0.20	2020-06-23	CRAN (R 4.0.2)
brew		2011-04-13	CRAN (R 4.0.0)
broom	0.7.1	2020-10-02	CRAN (R 4.0.2)
callr	3.4.4	2020-09-07	CRAN (R 4.0.2)
cellranger	1.1.0	2016-07-27	CRAN (R 4.0.0)
checkmate		2020-02-06	CRAN (R 4.0.0)
cli	2.0.2	2020-02-28	CRAN (R 4.0.0)
clipr		2019-07-23	CRAN (R 4.0.0)
colorspace	1.4-1	2019-03-18	CRAN (R 4.0.0)
commonmark		2018-12-01	CRAN (R 4.0.0)
covr		2020-09-16	CRAN (R 4.0.2)
cpp11		2020-10-01	CRAN (R 4.0.2)
crayon	1.3.4	2017-09-16	CRAN (R 4.0.0)
crosstalk		2020-03-13	CRAN (R 4.0.0)
curl		2019-12-02	CRAN (R 4.0.0)
DBI	1.1.0	2019-12-15	CRAN (R 4.0.0)
dbplyr	1.4.4	2020-05-27	CRAN (R 4.0.0)
desc	1.2.0	2018-05-01	CRAN (R 4.0.0)
devtools	2.3.2	2020-09-18	CRAN (R 4.0.2)
dichromat		2013-01-24	CRAN (R 4.0.2)
digest	0.6.25	2020-02-23	CRAN (R 4.0.0)
dplyr	1.0.2	2020-08-18	CRAN (R 4.0.2)
DT	0.15	2020-08-05	CRAN (R 4.0.2)
ellipsis	0.3.1	2020-05-15	CRAN (R 4.0.0)
evaluate	0.14	2019-05-28	CRAN (R 4.0.0)
fansi	0.4.1	2020-01-08	CRAN (R 4.0.0)
farver		2020-01-16	CRAN (R 4.0.0)
forcats	0.5.0	2020-03-01	CRAN (R 4.0.0)
fs	1.5.0	2020-07-31	CRAN (R 4.0.2)
gapminder		2017-10-31	CRAN (R 4.0.0)
gender			
genderdata			
generics	0.0.2	2018-11-29	CRAN (R 4.0.0)
geonames			
ggplot2	3.3.2	2020-06-19	CRAN (R 4.0.2)
gh		2020-01-24	CRAN (R 4.0.0)
git2r		2020-05-03	CRAN (R 4.0.0)
glue	1.4.2	2020-08-27	CRAN (R 4.0.2)
gridExtra		2017-09-09	CRAN (R 4.0.0)
gt		2020-08-05	CRAN (R 4.0.2)
gtable	0.3.0	2019-03-25	CRAN (R 4.0.0)
haven	2.3.1	2020-06-01	CRAN (R 4.0.0)
highr		2019-03-20	CRAN (R 4.0.0)
hms	0.5.3	2020-01-08	CRAN (R 4.0.0)
htmltools	0.5.0	2020-06-16	CRAN (R 4.0.2)
htmlwidgets	1.5.2	2020-10-03	CRAN (R 4.0.2)
httr	1.4.2	2020-07-20	CRAN (R 4.0.2)
ini		2018-05-20	CRAN (R 4.0.0)
isoband		2020-06-20	CRAN (R 4.0.2)
jsonlite	1.7.1	2020-09-07	CRAN (R 4.0.2)
knitr	1.30	2020-09-22	CRAN (R 4.0.2)
labeling		2014-08-23	CRAN (R 4.0.0)
later		2020-06-05	CRAN (R 4.0.0)
lattice		2020-04-02	CRAN (R 4.0.2)
lazyeval		2019-03-15	CRAN (R 4.0.0)
lifecycle	0.2.0	2020-03-06	CRAN (R 4.0.0)
lubridate	1.7.9	2020-06-08	CRAN (R 4.0.2)
magrittr	1.5	2014-11-22	CRAN (R 4.0.0)
markdown		2019-08-07	CRAN (R 4.0.0)
MASS		2020-04-26	CRAN (R 4.0.2)
Matrix		2019-11-27	CRAN (R 4.0.2)
memoise	1.1.0	2017-04-21	CRAN (R 4.0.0)
mgcv		2019-11-09	CRAN (R 4.0.2)
mime		2020-02-04	CRAN (R 4.0.0)
modelr	0.1.8	2020-05-19	CRAN (R 4.0.0)
munsell	0.5.0	2018-06-12	CRAN (R 4.0.0)
nlme		2020-05-24	CRAN (R 4.0.2)
openssl		2020-09-18	CRAN (R 4.0.2)
pillar	1.4.6	2020-07-10	CRAN (R 4.0.2)
pkgbuild	1.1.0	2020-07-13	CRAN (R 4.0.2)
pkgconfig	2.0.3	2019-09-22	CRAN (R 4.0.0)
pkgload	1.1.0	2020-05-29	CRAN (R 4.0.0)
praise		2015-08-11	CRAN (R 4.0.0)
prettyunits	1.1.1	2020-01-24	CRAN (R 4.0.0)
processx	3.4.4	2020-09-03	CRAN (R 4.0.2)
progress		2019-05-16	CRAN (R 4.0.0)
promises		2020-06-09	CRAN (R 4.0.2)
ps	1.3.4	2020-08-11	CRAN (R 4.0.2)
purrr	0.3.4	2020-04-17	CRAN (R 4.0.0)
R6	2.4.1	2019-11-12	CRAN (R 4.0.0)
rcmdcheck		2019-05-07	CRAN (R 4.0.0)
RColorBrewer		2014-12-07	CRAN (R 4.0.0)
Rcpp	1.0.5	2020-07-06	CRAN (R 4.0.2)
readr	1.3.1	2018-12-21	CRAN (R 4.0.2)
readxl	1.3.1	2019-03-13	CRAN (R 4.0.0)
rebird			
rematch		2016-04-21	CRAN (R 4.0.0)
rematch2		2020-05-01	CRAN (R 4.0.0)
remotes	2.2.0	2020-07-21	CRAN (R 4.0.2)
reprex	0.3.0	2019-05-16	CRAN (R 4.0.0)
rex		2020-04-21	CRAN (R 4.0.0)
rlang	0.4.7	2020-07-09	CRAN (R 4.0.2)
rmarkdown	2.4	2020-09-30	CRAN (R 4.0.2)
roxygen2		2020-06-27	CRAN (R 4.0.2)
rplos			
rprojroot	1.3-2	2018-01-03	CRAN (R 4.0.0)
rstudioapi	0.11	2020-02-07	CRAN (R 4.0.0)
rversions		2020-05-25	CRAN (R 4.0.0)
rvest	0.3.6	2020-07-25	CRAN (R 4.0.2)
sass		2020-03-18	CRAN (R 4.0.2)
scales	1.1.1	2020-05-11	CRAN (R 4.0.0)
selectr		2019-11-20	CRAN (R 4.0.0)
sessioninfo	1.1.1	2018-11-05	CRAN (R 4.0.0)
stringi	1.5.3	2020-09-09	CRAN (R 4.0.2)
stringr	1.4.0	2019-02-10	CRAN (R 4.0.0)
sys		2020-07-23	CRAN (R 4.0.2)
testthat	2.3.2	2020-03-02	CRAN (R 4.0.0)
tibble	3.0.3	2020-07-10	CRAN (R 4.0.2)
tidyr	1.1.2	2020-08-27	CRAN (R 4.0.2)
tidyselect	1.1.0	2020-05-11	CRAN (R 4.0.0)
tidyverse	1.3.0	2019-11-21	CRAN (R 4.0.0)
tinytex		2020-09-22	CRAN (R 4.0.2)
usethis	1.6.3	2020-09-17	CRAN (R 4.0.2)
utf8		2018-05-24	CRAN (R 4.0.0)
vctrs	0.3.4	2020-08-29	CRAN (R 4.0.2)
viridis		2018-03-29	CRAN (R 4.0.0)
viridisLite		2018-02-01	CRAN (R 4.0.0)
whisker		2019-08-28	CRAN (R 4.0.0)
withr	2.3.0	2020-09-22	CRAN (R 4.0.2)
xfun	0.18	2020-09-29	CRAN (R 4.0.2)
xml2	1.3.2	2020-04-23	CRAN (R 4.0.0)
xopen		2018-09-17	CRAN (R 4.0.0)
yaml	2.2.1	2020-02-01	CRAN (R 4.0.0)

License

Part I

Get your R act together

Chapter 1

Install R and RStudio

1.1 R and RStudio

- Install R, a free software environment for statistical computing and graphics from CRAN, the Comprehensive R Archive Network. I **highly recommend** you install a precompiled binary distribution for your operating system – use the links up at the top of the CRAN page linked above!
- Install RStudio's IDE (stands for *integrated development environment*), a powerful user interface for R. Get the Open Source Edition of RStudio Desktop.
  - I **highly recommend** you run the Preview version. I find these quite stable and you'll get the cool new features! Update to new Preview versions often.
  - Of course, there are also official releases available here.
  - RStudio comes with a **text editor**, so there is no immediate need to install a separate stand-alone editor.
  - RStudio can **interface with Git(Hub)**. However, you must do all the Git(Hub) set up described elsewhere (see Happy Git and GitHub for the useR) before you can take advantage of this.

If you have a pre-existing installation of R and/or RStudio, we **highly recommend** that you reinstall both and get as current as possible. It can be considerably harder to run old software than new.

- If you upgrade R, you will need to update any packages you have installed. The command below should get you started, though you may need to specify more arguments if, e.g., you have been using a non-default library for your packages.

```
update.packages(ask = FALSE, checkBuilt = TRUE)
```

Note: this will only look for updates on CRAN. So if you use a package that lives *only* on GitHub or if you want a development version from GitHub, you will need to update manually, e.g. via `devtools::install_github()`.

1.2 Testing testing

- Do whatever is appropriate for your OS to launch RStudio. You should get a window similar to the screenshot you see here, but yours will be more boring because you haven't written any code or made any figures yet!

- Put your cursor in the pane labelled Console, which is where you interact with the live R process. Create a simple object with code like `x <- 2 * 4` (followed by enter or return). Then inspect the `x` object by typing `x` followed by enter or return. You should see the value 8 print to screen. If yes, you've succeeded in installing R and RStudio.

1.3 Add-on packages

R is an extensible system and many people share useful code they have developed as a *package* via CRAN and GitHub. To install a package from CRAN, for example the dplyr package for data manipulation, here is one way to do it in the R console (there are others).

```
install.packages("dplyr", dependencies = TRUE)
```

By including `dependencies = TRUE`, we are being explicit and extra-careful to install any additional packages the target package, dplyr in the example above, needs to have around.

You could use the above method to install the following packages, all of which we will use:

- tidyr
- ggplot2

1.4 Further resources

The above will get your basic setup ready but here are some links if you are interested in reading a bit further.

- How to Use RStudio
- RStudio's leads for learning R
- R FAQ
- R Installation and Administration
- More about add-on packages in the R Installation and Administration Manual

Chapter 2

R basics and workflows

2.1 Basics of working with R at the command line and RStudio goodies

Launch RStudio/R.

Notice the default panes:

- Console (entire left)
- Environment / History (tabbed in upper right)
- Files / Plots / Packages / Help (tabbed in lower right)

FYI: you can change the default location of the panes, among many other things: Customizing RStudio.

Go into the Console, where we interact with the live R process.

Make an assignment and then inspect the object you just created:

```
x <- 3 * 4
x
#> [1] 12
```

All R statements where you create objects – "assignments" – have this form:

```
objectName <- value
```

and in my head I hear, e.g., "x gets 12".

You will make lots of assignments and the operator `<-` is a pain to type. Don't be lazy and use `=`, although it would work, because it will just sow confusion later. Instead, utilize RStudio's keyboard shortcut: Alt + - (the minus sign).

Notice that RStudio automagically surrounds `<-` with spaces, which demonstrates a useful code formatting practice. Code is miserable to read on a good day. Give your eyes a break and use spaces.

RStudio offers many handy keyboard shortcuts. Also, Alt+Shift+K brings up a keyboard shortcut reference card.

Object names cannot start with a digit and cannot contain certain other characters such as a comma or a space. You will be wise to adopt a convention for demarcating words in names.

```
i_use_snake_case
other.people.use.periods
evenOthersUseCamelCase
```

Make another assignment:

```
this_is_a_really_long_name <- 2.5
```

To inspect this, try out RStudio's completion facility: type the first few characters, press TAB, add characters until you disambiguate, then press return.

Make another assignment:

```
jenny_rocks <- 2 ^ 3
```

Let's try to inspect:

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

Implicit contract with the computer / scripting language: Computer will do tedious computation for you. In return, you will be completely precise in your instructions. Typos matter. Case matters. Get better at typing.

R has a mind-blowing collection of built-in functions that are accessed like so:

```
functionName(arg1 = val1, arg2 = val2, and so on)
```

Let's try using `seq()` which makes regular sequences of numbers and, while we're at it, demo more helpful features of RStudio.

Type `se` and hit TAB. A pop up shows you possible completions. Specify `seq()` by typing more to disambiguate or using the up/down arrows to select. Notice the floating tool-tip-type help that pops up, reminding you of a function's arguments. If you want even more help, press F1 as directed to get the full documentation in the help tab of the lower right pane. Now open the parentheses and notice the automatic addition of the closing parenthesis and the placement of cursor in the middle. Type the arguments `1, 10` and hit return. RStudio also exits the parenthetical expression for you. IDEs are great.

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

The above also demonstrates something about how R resolves function arguments. You can always specify in `name = value` form. But if you do not, R attempts to resolve by position. So above, it is assumed that we want a sequence `from = 1` that goes `to = 10`. Since we didn't specify step size, the default value of `by` in the function definition is used, which ends up being 1 in this case. For functions I call often, I might use this resolve by position for the first argument or maybe the first two. After that, I always use `name = value`.

Make this assignment and notice similar help with quotation marks.

```
yo <- "hello world"
```

If you just make an assignment, you don't get to see the value, so then you're tempted to immediately inspect.

```
y <- seq(1, 10)
y
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen.

```
(y <- seq(1, 10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Not all functions have (or require) arguments:

```
date()
#> [1] "Tue Oct 27 17:42:05 2020"
```

Now look at your workspace – in the upper right pane. The workspace is where user-defined objects accumulate. You can also get a listing of these objects with commands:

```
objects()
#> [1] "check_quietly"                 "install_quietly"
#> [3] "jenny_rocks"                   "pretty_install"
#> [5] "shhh_check"                    "this_is_a_really_long_name"
#> [7] "x"                             "y"
#> [9] "yo"
ls()
#> [1] "check_quietly"                 "install_quietly"
#> [3] "jenny_rocks"                   "pretty_install"
#> [5] "shhh_check"                    "this_is_a_really_long_name"
#> [7] "x"                             "y"
#> [9] "yo"
```

If you want to remove the object named `y`, you can do this:

```
rm(y)
```

To remove everything:

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

or click the broom in RStudio's Environment pane.

2.2 Workspace and working directory

One day you will need to quit R, go do something else and return to your analysis later.

One day you will have multiple analyses going that use R and you want to keep them separate.

One day you will need to bring data from the outside world into R and send numerical results and figures from R back out into the world.

To handle these real life situations, you need to make two decisions:

- What about your analysis is "real", i.e. will you save it as your lasting record of what happened?
- Where does your analysis "live"?

2.2.1 Workspace, `.RData`

As a beginning R user, it's OK to consider your workspace "real". *Very soon*, I urge you to evolve to the next level, where you consider your saved R scripts as "real". (In either case, of course the input data is very much real and requires preservation!) With the input data and the R code you used, you can reproduce *everything*. You can make your analysis fancier. You can get to the bottom of puzzling results and discover and fix bugs in your code. You can reuse the code to conduct similar analyses in new projects. You can remake a figure with different aspect ratio or save is as TIFF instead of PDF. You are ready to take questions. You are ready for the future.

If you regard your workspace as "real" (saving and reloading all the time), if you need to redo analysis … you're going to either redo a lot of typing (making mistakes all the way) or will have to mine your R history for the commands you used. Rather than becoming an expert on managing the R history, a better use of your time and psychic energy is to keep your "good" R code in a script for future reuse.

Because it can be useful sometimes, note the commands you've recently run appear in the History pane.

But you don't have to choose right now and the two strategies are not incompatible. Let's demo the save / reload the workspace approach.

Upon quitting R, you have to decide if you want to save your workspace, for potential restoration the next time you launch R. Depending on your set up, R or your IDE, e.g. RStudio, will probably prompt you to make this decision.

Quit R/RStudio, either from the menu, using a keyboard shortcut, or by typing `q()` in the Console. You'll get a prompt like this:

> Save workspace image to ~/.Rdata?

Note where the workspace image is to be saved and then click "Save".

Using your favorite method, visit the directory where image was saved and verify there is a file named `.RData`. You will also see a file `.Rhistory`, holding the commands submitted in your recent session.

Restart RStudio. In the Console you will see a line like this:

```
[Workspace loaded from ~/.RData]
```

indicating that your workspace has been restored. Look in the Workspace pane and you'll see the same objects as before. In the History tab of the same pane, you should also see your command history. You're back in business. This way of starting and stopping analytical work will not serve you well for long but it's a start.

2.2.2 Working directory

Any process running on your computer has a notion of its "working directory". In R, this is where R will look, by default, for files you ask it to load. It also where, by default, any files you write to disk will go. Chances are your current working directory is the directory we inspected above, i.e. the one where RStudio wanted to save the workspace.

You can explicitly check your working directory with:

```
getwd()
```

It is also displayed at the top of the RStudio console.

As a beginning R user, it's OK let your home directory or any other weird directory on your computer be R's working directory. *Very soon*, I urge you to evolve to the next level, where you organize your analytical projects into directories and, when working on project A, set R's working directory to the associated directory.

Although I do not recommend it, in case you're curious, you can set R's working directory at the command line like so:

```
setwd("~/myCoolProject")
```

Although I do not recommend it, you can also use RStudio's Files pane to navigate to a directory and then set it as working directory from the menu: *Session > Set Working Directory > To Files Pane Location*. (You'll see even more options there). Or within the Files pane, choose "More" and "Set As Working Directory".

But there's a better way. A way that also puts you on the path to managing your R work like an expert.

2.3 RStudio projects

Keeping all the files associated with a project organized together – input data, R scripts, analytical results, figures – is such a wise and common practice that RStudio has built-in support for this via its *projects*.

Let's make one to use for the rest of this workshop/class. Do this: *File > New Project…*. The directory name you choose here will be the project name. Call it whatever you want (or follow me for convenience).

I created a directory and, therefore RStudio project, called `swc` in my `tmp` directory, FYI.

```
setwd("~/tmp/swc")
```

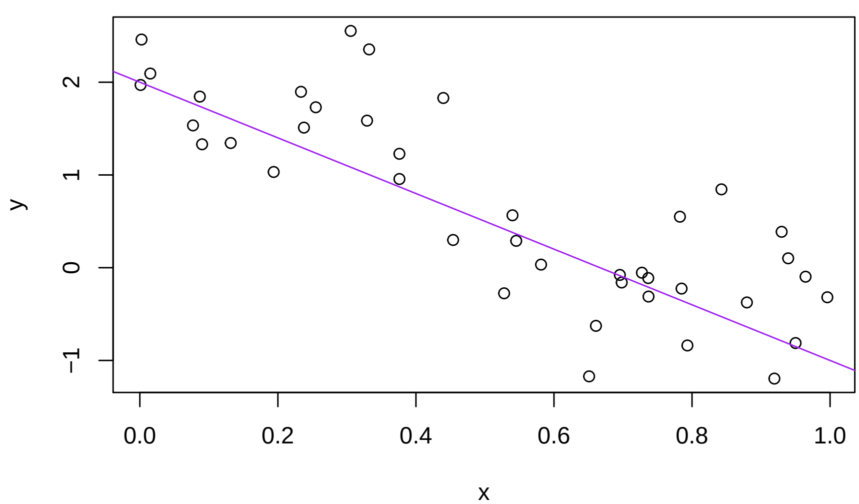
Now check that the "home" directory for your project is the working directory of our current R process:

```
getwd()
```

I can't print my output here because this document itself does not reside in the RStudio Project we just created.

Let's enter a few commands in the Console, as if we are just beginning a project:

```
a <- 2
b <- -3
sig_sq <- 0.5
x <- runif(40)
y <- a + b * x + rnorm(40, sd = sqrt(sig_sq))
(avg_x <- mean(x))
#> [1] 0.52
write(avg_x, "avg_x.txt")
plot(x, y)
abline(a, b, col = "purple")
```



```
dev.print(pdf, "toy_line_plot.pdf")
#> pdf
#>   2
```

Let's say this is a good start of an analysis and your ready to start preserving the logic and code. Visit the History tab of the upper right pane. Select these commands. Click "To Source". Now you have a new pane containing a nascent R script. Click on the floppy disk to save. Give it a name ending in `.R` or `.r`, I used `toy-line.r` and note that, by default, it will go in the directory associated with your project.

Quit RStudio. Inspect the folder associated with your project if you wish. Maybe view the PDF in an external viewer.

Restart RStudio. Notice that things, by default, restore to where we were earlier, e.g. objects in the workspace, the command history, which files are open for editing, where we are in the file system browser, the working directory for the R process, etc. These are all Good Things.

Change some things about your code. Top priority would be to set a sample size `n` at the top, e.g. `n <- 40`, and then replace all the hard-wired 40's with `n`. Change some other minor-but-detectable stuff, e.g. alter the sample size `n`, the slope of the line `b`,the color of the line … whatever. Practice the different ways to re-run the code:

- Walk through line by line by keyboard shortcut (Command+Enter) or mouse (click "Run" in the upper right corner of editor pane).
- Source the entire document – equivalent to entering `source('toy-line.r')` in the Console – by keyboard shortcut (Shift+Command+S) or mouse (click "Source" in the upper right corner of editor pane or select from the mini-menu accessible from the associated down triangle).
- Source with echo from the Source mini-menu.

Visit your figure in an external viewer to verify that the PDF is changing as you expect.

In your favorite OS-specific way, search your files for `toy_line_plot.pdf` and presumably you will find the PDF itself (no surprise) but *also the script that created it (`toy-line.r`)*. This latter phenomenon is a huge win. One day you will want to remake a figure or just simply understand where it came from. If you rigorously save figures to file **with R code and not ever ever ever the mouse or the clipboard**, you will sing my praises one day. Trust me.

2.4 Stuff

It is traditional to save R scripts with a `.R` or `.r` suffix. Follow this convention unless you have some extraordinary reason not to.

Comments start with one or more `#` symbols. Use them. RStudio helps you (de)comment selected lines with Ctrl+Shift+C (Windows and Linux) or Command+Shift+C (Mac).

Clean out the workspace, i.e. pretend like you've just revisited this project after a long absence. The broom icon or `rm(list = ls())`. Good idea to do this, restart R (available from the Session menu), re-run your analysis to truly check that the code you're saving is complete and correct (or at least rule out obvious problems!).

This workflow will serve you well in the future:

- Create an RStudio project for an analytical project
- Keep inputs there (we'll soon talk about importing)
- Keep scripts there; edit them, run them in bits or as a whole from there
- Keep outputs there (like the PDF written above)

Avoid using the mouse for pieces of your analytical workflow, such as loading a dataset or saving a figure. Terribly important for reproducibility and for making it possible to retrospectively determine how a numerical table or PDF was actually produced (searching on local disk on filename, among `.R` files, will lead to the relevant script).

Many long-time users never save the workspace, never save `.RData` files (I'm one of them), never save or consult the history. Once/if you get to that point, there are options available in RStudio to disable the loading of `.RData` and permanently suppress the prompt on exit to save the workspace (go to *Tools > Options > General*).

For the record, when loading data into R and/or writing outputs to file, you can always specify the absolute path and thereby insulate yourself from the current working directory. This is rarely necessary when using RStudio projects properly.

Part II

Version control and R Markdown

Overview

Although this part now links out to external resources, if you're working through this material on your own, let this be a nudge to pause around here and think about your workflow. I give you permission to spend some time and energy sorting this out! It can be as or more important than learning a new R function or package. The experts don't talk about this much, because they've already got a workflow and it's something they do almost without thinking.

Working through subsequent material in R Markdown documents, possibly using Git and GitHub to track and share your progress, is a great idea and will leave you more prepared for your future data analysis projects. Typing individual lines of R code is but a small part of data analysis and it pays off to think holistically about your workflow.

Chapter 3

Git, GitHub, and RStudio

At this point in STAT 545, all students receive their own STAT 545 GitHub repository that they will use to develop their course work throughout the rest of the course.

This has two purposes:

- It is helpful for course mechanics, e.g. homework submission and grading, peer review.
- Learning to use Git and GitHub, with R and RStudio, is a legitimate pedagogical goal.

Our instructions around installation, setup, and early Git usage eventually grew so extensive that we created a dedicated website. This content can now be found here:

https://happygitwithr.com

Chapter 4

R Markdown

STAT 545 course work is generally submitted in the form of R Markdown documents. Students submit an `.Rmd` file, which they have executed or rendered to a `.md` markdown file. R Markdown is a very accessible way to create computational documents that combine prose and tables and figures produced by R code.

An introductory R Markdown workflow, including how it intersects with Git, GitHub, and RStudio, is now maintained within the Happy Git site:

Test drive R Markdown

Part III

Data analysis 1

Chapter 5

Basic care and feeding of data in R

5.1 Buckle your seatbelt

Ignore if you don't need this bit of support.

Now is the time to make sure you are working in an appropriate directory on your computer, probably through the use of an RStudio project. Enter `getwd()` in the Console to see current working directory or, in RStudio, this is displayed in the bar at the top of Console.

You should clean out your workspace. In RStudio, click on the "Clear" broom icon from the Environment tab or use *Session > Clear Workspace*. You can also enter `rm(list = ls())` in the Console to accomplish same.

Now restart R. This will ensure you don't have any packages loaded from previous calls to `library()`. In RStudio, use *Session > Restart R*. Otherwise, quit R with `q()` and re-launch it.

Why do we do this? So that the code you write is complete and re-runnable. If you return to a clean slate often, you will root out hidden dependencies where one snippet of code only works because it relies on objects created by code saved elsewhere or, much worse, never saved at all. Similarly, an aggressive clean slate approach will expose any usage of packages that have not been explicitly loaded.

Finally, open a new R script and develop and run your code from there. In RStudio, use *File > New File > R Script*. Save this script with a name ending in `.r` or `.R`, containing no spaces or other funny stuff, and that evokes whatever it is we're doing today. Example: `cm004_data-care-feeding.r`.

Another great idea is to do this in an R Markdown document. See Test drive R Markdown for a refresher.

5.2 Data frames are awesome

Whenever you have rectangular, spreadsheet-y data, your default data receptacle in R is a data frame. Do not depart from this without good reason. Data frames are awesome because…

- Data frames package related variables neatly together,
  - keeping them in sync vis-a-vis row order
  - applying any filtering of observations uniformly
- Most functions for inference, modelling, and graphing are happy to be passed a data frame via a `data =` argument. This has been true in base R for a long time.
- The set of packages known as the tidyverse takes this one step further and explicitly prioritizes the processing of data frames. This includes popular packages like dplyr and ggplot2. In fact the tidyverse prioritizes a special flavor of data frame, called a "tibble".

Data frames – unlike general arrays or, specifically, matrices in R – can hold variables of different flavors, such as character data (subject ID or name), quantitative data (white blood cell count), and categorical information (treated vs. untreated). If you use homogeneous structures, like matrices, for data analysis, you are likely to make the terrible mistake of spreading a dataset out over multiple, unlinked objects. Why? Because you can't put character data, such as subject name, into the numeric matrix that holds white blood cell count. This fragmentation is a Bad Idea.

5.3 Get the Gapminder data

We will work with some of the data from the Gapminder project. I've released this as an R package called gapminder, so we can install it from CRAN like so:

```
install.packages("gapminder")
```

Now load the package:

```
library(gapminder)
```

5.4 Meet the `gapminder` data frame or "tibble"

By loading the gapminder package, we now have access to a data frame by the same name. Get an overview of this with `str()`, which displays the structure of an object.

```
str(gapminder)
#> tibble [1,704 × 6] (S3: tbl_df/tbl/data.frame)
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 ..
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 ..
#>  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 ..
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
#>  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 130794..
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

`str()` will provide a sensible description of almost anything and, worst case, nothing bad can actually happen. When in doubt, just `str()` some of the recently created objects to get some ideas about what to do next.

We could print the `gapminder` object itself to screen. However, if you've used R before, you might be reluctant to do this, because large datasets just fill up your Console and provide very little insight.

This is the first big win for **tibbles**. The tidyverse offers a special case of R's default data frame: the "tibble", which is a nod to the actual class of these objects, `tbl_df`.

If you have not already done so, install the tidyverse meta-package now:

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

Now load it:

```
library(tidyverse)
#>  Attaching packages                 tidyverse 1.3.0
#>  ggplot2 3.3.2      purrr   0.3.4
#>  tibble  3.0.3      dplyr   1.0.2
#>  tidyr   1.1.2      stringr 1.4.0
#>  readr   1.3.1      forcats 0.5.0
#>  Conflicts                   tidyverse_conflicts()
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Now we can boldly print `gapminder` to screen! It is a tibble (and also a regular data frame) and the tidyverse provides a nice print method that shows the most important stuff and doesn't fill up your Console.

```
## see? it's still a regular data frame, but also a tibble
class(gapminder)
#> [1] "tbl_df"     "tbl"        "data.frame"
gapminder
```

```
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # … with 1,694 more rows
```

If you are dealing with plain vanilla data frames, you can rein in data frame printing explicitly with `head()` and `tail()`. Or turn it into a tibble with `as_tibble()`!

```
head(gapminder)
#> # A tibble: 6 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
tail(gapminder)
#> # A tibble: 6 x 6
#>   country  continent  year lifeExp      pop gdpPercap
#>   <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Zimbabwe Africa     1982    60.4  7636524      789.
#> 2 Zimbabwe Africa     1987    62.4  9216418      706.
#> 3 Zimbabwe Africa     1992    60.4 10704340      693.
#> 4 Zimbabwe Africa     1997    46.8 11404948      792.
#> 5 Zimbabwe Africa     2002    40.0 11926563      672.
#> 6 Zimbabwe Africa     2007    43.5 12311143      470.
as_tibble(iris)
#> # A tibble: 150 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
```

```
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # … with 140 more rows
```

More ways to query basic info on a data frame:

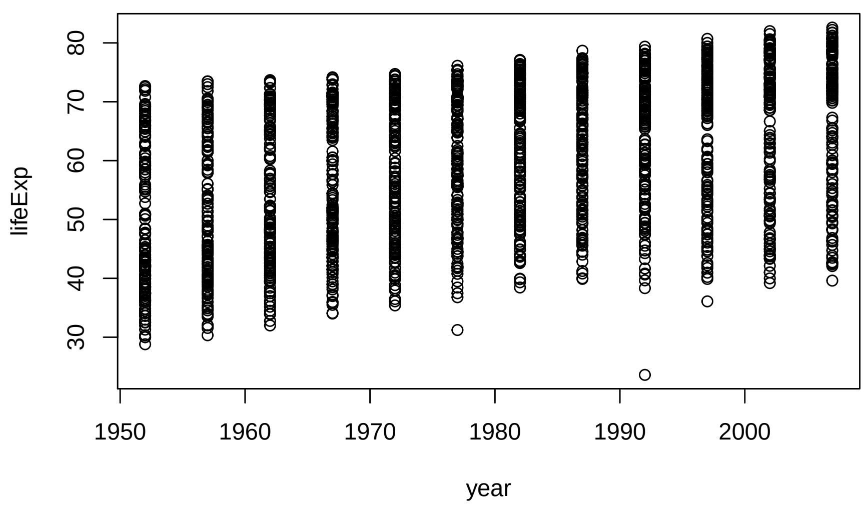
```
names(gapminder)
#> [1] "country"   "continent" "year"      "lifeExp"   "pop"
#> [6] "gdpPercap"
ncol(gapminder)
#> [1] 6
length(gapminder)
#> [1] 6
dim(gapminder)
#> [1] 1704    6
nrow(gapminder)
#> [1] 1704
```

A statistical overview can be obtained with `summary()`:

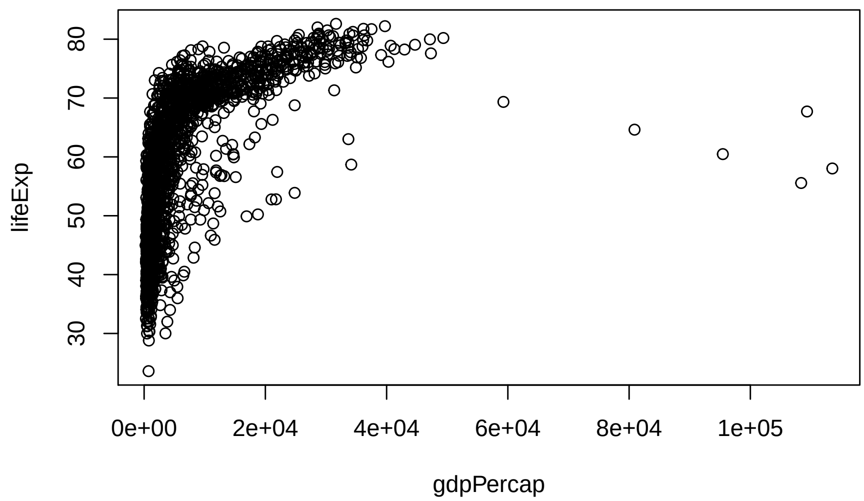
```
summary(gapminder)
#>         country        continent        year         lifeExp
#>  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.6
#>  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.2
#>  Algeria    :  12   Asia    :396   Median :1980   Median :60.7
#>  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.5
#>  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.8
#>  Australia  :  12                  Max.   :2007   Max.   :82.6
#>  (Other)    :1632
#>       pop              gdpPercap
#>  Min.   :6.00e+04   Min.   :   241
#>  1st Qu.:2.79e+06   1st Qu.:  1202
#>  Median :7.02e+06   Median :  3532
#>  Mean   :2.96e+07   Mean   :  7215
#>  3rd Qu.:1.96e+07   3rd Qu.:  9325
#>  Max.   :1.32e+09   Max.   :113523
#>
```

Although we haven't begun our formal coverage of visualization yet, it's so important for smell-testing dataset that we will make a few figures anyway. Here we use only base R graphics, which are very basic.

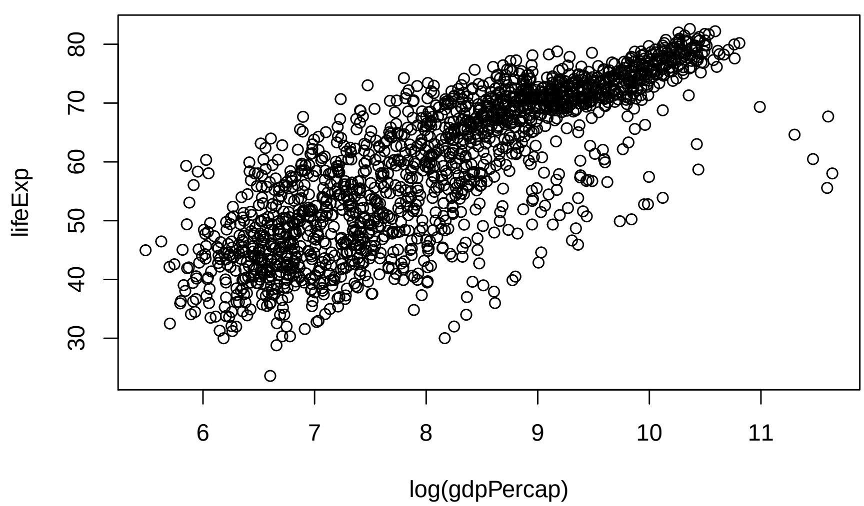
```
plot(lifeExp ~ year, gapminder)
```




```
plot(lifeExp ~ gdpPercap, gapminder)
```




```
plot(lifeExp ~ log(gdpPercap), gapminder)
```



Let's go back to the result of `str()` to talk about what a data frame is.

```
str(gapminder)
#> tibble [1,704 × 6] (S3: tbl_df/tbl/data.frame)
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 ..
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 ..
#>  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 ..
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
#>  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 130794..
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

A data frame is a special case of a *list*, which is used in R to hold just about anything. Data frames are a special case where the length of each list component is the same. Data frames are superior to matrices in R because they can hold vectors of different flavors, e.g. numeric, character, and categorical data can be stored together. This comes up a lot!

5.5 Look at the variables inside a data frame

To specify a single variable from a data frame, use the dollar sign `$`. Let's explore the numeric variable for life expectancy.

```
head(gapminder$lifeExp)
#> [1] 28.8 30.3 32.0 34.0 36.1 38.4
summary(gapminder$lifeExp)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    23.6    48.2    60.7    59.5    70.8    82.6
hist(gapminder$lifeExp)
```

The year variable is an integer variable, but since there are so few unique values it also functions a bit like a categorical variable.

```
summary(gapminder$year)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1952    1966    1980    1980    1993    2007
table(gapminder$year)
#> 
#> 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
#>  142  142  142  142  142  142  142  142  142  142  142  142
```

The variables for country and continent hold truly categorical information, which is stored as a *factor* in R.

```
class(gapminder$continent)
#> [1] "factor"
summary(gapminder$continent)
#>   Africa Americas     Asia   Europe  Oceania
#>      624      300      396      360       24
levels(gapminder$continent)
#> [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
nlevels(gapminder$continent)
#> [1] 5
```

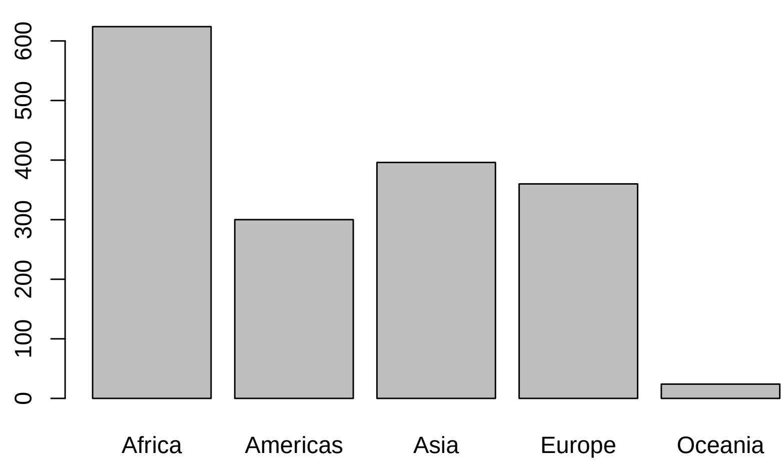
The **levels** of the factor `continent` are "Africa", "Americas", etc. and this is what's usually presented to your eyeballs by R. In general, the levels are friendly human-readable character strings, like "male/female" and "control/treated". But *never ever ever* forget that, under the hood, R is really storing integer codes 1, 2, 3, etc. Look at the result from `str(gapminder$continent)` if you are skeptical.

```
str(gapminder$continent)
#>  Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
```

This Janus-like nature of factors means they are rich with booby traps for the unsuspecting but they are a necessary evil. I recommend you resolve to learn how to properly care and feed for factors. The pros far outweigh the cons. Specifically in modelling and figure-making, factors are anticipated and accommodated by the functions and packages you will want to exploit.

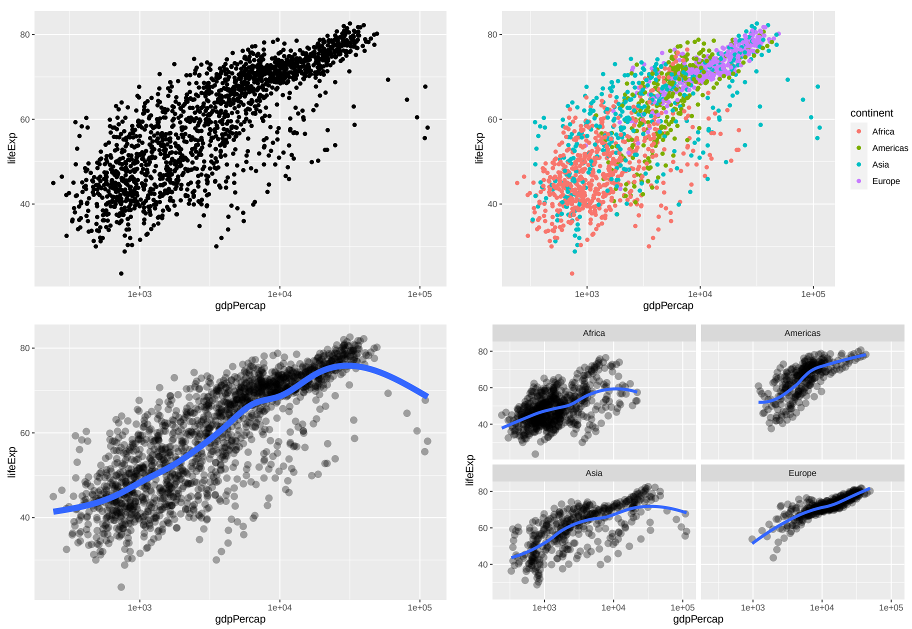
Here we count how many observations are associated with each continent and, as usual, try to portray that info visually. This makes it much easier to quickly see that African countries are well represented in this dataset.

```
table(gapminder$continent)
#> 
#>   Africa Americas     Asia   Europe  Oceania 
#>      624      300      396      360       24
barplot(table(gapminder$continent))
```



In the figures below, we see how factors can be put to work in figures. The `continent` factor is easily mapped into "facets" or colors and a legend by the ggplot2 package. *Making figures with ggplot2 is covered in Chapter* **??** *so feel free to just sit back and enjoy these plots or blindly copy/paste.*

```
## we exploit the fact that ggplot2 was installed and loaded via the tidyverse
p <- ggplot(filter(gapminder, continent != "Oceania"),
            aes(x = gdpPercap, y = lifeExp)) # just initializes
p <- p + scale_x_log10() # log the x axis the right way
p + geom_point() # scatterplot
p + geom_point(aes(color = continent)) # map continent to color
p + geom_point(alpha = (1/3), size = 3) + geom_smooth(lwd = 3, se = FALSE)
#> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
p + geom_point(alpha = (1/3), size = 3) + facet_wrap(~ continent) +
  geom_smooth(lwd = 1.5, se = FALSE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



5.6 Recap

- Use data frames!!!
- Use the tidyverse!!! This will provide a special type of data frame called a "tibble" that has nice default printing behavior, among other benefits.

- When in doubt, `str()` something or print something.
- Always understand the basic extent of your data frames: number of rows and columns.
- Understand what flavor the variables are.
- Use factors!!! But with intention and care.
- Do basic statistical and visual sanity checking of each variable.
- Refer to variables by name, e.g., `gapminder$lifeExp`, not by column number. Your code will be more robust and readable.

Chapter 6

Introduction to dplyr

6.1 Intro

dplyr is a package for data manipulation, developed by Hadley Wickham and Romain Francois. It is built to be fast, highly expressive, and open-minded about how your data is stored. It is installed as part of the tidyverse meta-package and, as a core package, it is among those loaded via `library(tidyverse)`.

dplyr's roots are in an earlier package called plyr, which implements the "split-apply-combine" strategy for data analysis (Wickham, 2011). Where plyr covers a diverse set of inputs and outputs (e.g., arrays, data frames, lists), dplyr has a laser-like focus on data frames or, in the tidyverse, "tibbles". dplyr is a package-level treatment of the `ddply()` function from plyr, because "data frame in, data frame out" proved to be so incredibly important.

Have no idea what I'm talking about? Not sure if you care? If you use these base R functions: `subset()`, `apply()`, `[sl]apply()`, `tapply()`, `aggregate()`, `split()`, `do.call()`, `with()`, `within()`, then you should keep reading. Also, if you use `for()` loops a lot, you might enjoy learning other ways to iterate over rows or groups of rows or variables in a data frame.

6.1.1 Load dplyr and gapminder

I choose to load the tidyverse, which will load dplyr, among other packages we use incidentally below.

```
library(tidyverse)
#>   Attaching packages                          tidyverse 1.3.0
#>  ggplot2 3.3.2       purrr   0.3.4
```

```
#>  tibble  3.0.3      dplyr   1.0.2
#>  tidyr   1.1.2      stringr 1.4.0
#>  readr   1.3.1      forcats 0.5.0
#>  Conflicts                  tidyverse_conflicts()
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Also load gapminder.

```
library(gapminder)
```

6.1.2 Say hello to the `gapminder` tibble

The `gapminder` data frame is a special kind of data frame: a tibble.

```
gapminder
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # … with 1,694 more rows
```

It's tibble-ness is why we get nice compact printing. For a reminder of the problems with base data frame printing, go type `iris` in the R Console or, better yet, print a data frame to screen that has lots of columns.

Note how `gapminder`'s `class()` includes `tbl_df`; the "tibble" terminology is a nod to this.

```
class(gapminder)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

There will be some functions, like `print()`, that know about tibbles and do something special. There will others that do not, like `summary()`. In which case the regular data frame treatment will happen, because every tibble is also a regular data frame.

To turn any data frame into a tibble use `as_tibble()`:

```
as_tibble(iris)
#> # A tibble: 150 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # … with 140 more rows
```

6.2 Think before you create excerpts of your data …

If you feel the urge to store a little snippet of your data:

```
(canada <- gapminder[241:252, ])
#> # A tibble: 12 x 6
#>    country continent  year lifeExp      pop gdpPercap
#>    <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Canada  Americas   1952    68.8 14785584    11367.
#>  2 Canada  Americas   1957    70.0 17010154    12490.
#>  3 Canada  Americas   1962    71.3 18985849    13462.
#>  4 Canada  Americas   1967    72.1 20819767    16077.
#>  5 Canada  Americas   1972    72.9 22284500    18971.
#>  6 Canada  Americas   1977    74.2 23796400    22091.
#>  7 Canada  Americas   1982    75.8 25201900    22899.
#>  8 Canada  Americas   1987    76.9 26549700    26627.
#>  9 Canada  Americas   1992    78.0 28523502    26343.
#> 10 Canada  Americas   1997    78.6 30305843    28955.
#> 11 Canada  Americas   2002    79.8 31902268    33329.
#> 12 Canada  Americas   2007    80.7 33390141    36319.
```

Stop and ask yourself …

> Do I want to create mini datasets for each level of some factor (or unique combination of several factors) … in order to compute or graph something?

If YES, **use proper data aggregation techniques** or faceting in ggplot2 – **don't subset the data**. Or, more realistic, only subset the data as a temporary measure while you develop your elegant code for computing on or visualizing these data subsets.

If NO, then maybe you really do need to store a copy of a subset of the data. But seriously consider whether you can achieve your goals by simply using the `subset =` argument of, e.g., the `lm()` function, to limit computation to your excerpt of choice. Lots of functions offer a `subset =` argument!

Copies and excerpts of your data clutter your workspace, invite mistakes, and sow general confusion. Avoid whenever possible.

Reality can also lie somewhere in between. You will find the workflows presented below can help you accomplish your goals with minimal creation of temporary, intermediate objects.

6.3 Use `filter()` to subset data row-wise

`filter()` takes logical expressions and returns the rows for which all are `TRUE`.

```
filter(gapminder, lifeExp < 29)
#> # A tibble: 2 x 6
#>   country     continent  year lifeExp     pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8 8425333      779.
#> 2 Rwanda      Africa     1992    23.6 7290203      737.
filter(gapminder, country == "Rwanda", year > 1979)
#> # A tibble: 6 x 6
#>   country continent  year lifeExp     pop gdpPercap
#>   <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
#> 1 Rwanda  Africa     1982    46.2 5507565      882.
#> 2 Rwanda  Africa     1987    44.0 6349365      848.
#> 3 Rwanda  Africa     1992    23.6 7290203      737.
#> 4 Rwanda  Africa     1997    36.1 7212583      590.
#> 5 Rwanda  Africa     2002    43.4 7852401      786.
#> 6 Rwanda  Africa     2007    46.2 8860588      863.
filter(gapminder, country %in% c("Rwanda", "Afghanistan"))
#> # A tibble: 24 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
```

```
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # … with 14 more rows
```

Compare with some base R code to accomplish the same things:

```
gapminder[gapminder$lifeExp < 29, ] ## repeat `gapminder`, [i, j] indexing is distracting
subset(gapminder, country == "Rwanda") ## almost same as filter; quite nice actually
```

Under no circumstances should you subset your data the way I did at first:

```
excerpt <- gapminder[241:252, ]
```

Why is this a terrible idea?

- It is not self-documenting. What is so special about rows 241 through 252?
- It is fragile. This line of code will produce different results if someone changes the row order of `gapminder`, e.g. sorts the data earlier in the script.

```
filter(gapminder, country == "Canada")
```

This call explains itself and is fairly robust.

6.4 Meet the new pipe operator

Before we go any further, we should exploit the new pipe operator that the tidyverse imports from the magrittr package by Stefan Bache. This is going to change your data analytical life. You no longer need to enact multi-operation commands by nesting them inside each other, like so many Russian nesting dolls. This new syntax leads to code that is much easier to write and to read.

Here's what it looks like: `%>%`. The RStudio keyboard shortcut: Ctrl+Shift+M (Windows), Cmd+Shift+M (Mac).

Let's demo then I'll explain.

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

This is equivalent to `head(gapminder)`. The pipe operator takes the thing on the left-hand-side and **pipes** it into the function call on the right-hand-side – literally, drops it in as the first argument.

Never fear, you can still specify other arguments to this function! To see the first 3 rows of `gapminder`, we could say `head(gapminder, 3)` or this:

```
gapminder %>% head(3)
#> # A tibble: 3 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
```

I've advised you to think "gets" whenever you see the assignment operator, `<-`. Similarly, you should think "then" whenever you see the pipe operator, `%>%`.

You are probably not impressed yet, but the magic will soon happen.

6.5 Use `select()` to subset the data on variables or columns.

Back to dplyr....

Use `select()` to subset the data on variables or columns. Here's a conventional call:

```
select(gapminder, year, lifeExp)
#> # A tibble: 1,704 x 2
#>     year lifeExp
```

```
#>    <int>   <dbl>
#>  1  1952    28.8
#>  2  1957    30.3
#>  3  1962    32.0
#>  4  1967    34.0
#>  5  1972    36.1
#>  6  1977    38.4
#>  7  1982    39.9
#>  8  1987    40.8
#>  9  1992    41.7
#> 10  1997    41.8
#> # … with 1,694 more rows
```

And here's the same operation, but written with the pipe operator and piped through `head()`:

```
gapminder %>%
  select(year, lifeExp) %>%
  head(4)
#> # A tibble: 4 x 2
#>    year lifeExp
#>   <int>   <dbl>
#> 1  1952    28.8
#> 2  1957    30.3
#> 3  1962    32.0
#> 4  1967    34.0
```

Think: "Take `gapminder`, then select the variables `year` and `lifeExp`, then show the first 4 rows."

6.6 Revel in the convenience

Here's the data for Cambodia, but only certain variables:

```
gapminder %>%
  filter(country == "Cambodia") %>%
  select(year, lifeExp)
#> # A tibble: 12 x 2
#>     year lifeExp
#>    <int>   <dbl>
#>  1  1952    39.4
#>  2  1957    41.4
#>  3  1962    43.4
```

```
#>  4  1967    45.4
#>  5  1972    40.3
#>  6  1977    31.2
#>  7  1982    51.0
#>  8  1987    53.9
#>  9  1992    55.8
#> 10  1997    56.5
#> 11  2002    56.8
#> 12  2007    59.7
```

and what a typical base R call would look like:

```
gapminder[gapminder$country == "Cambodia", c("year", "lifeExp")]
#> # A tibble: 12 x 2
#>     year lifeExp
#>    <int>   <dbl>
#>  1  1952    39.4
#>  2  1957    41.4
#>  3  1962    43.4
#>  4  1967    45.4
#>  5  1972    40.3
#>  6  1977    31.2
#>  7  1982    51.0
#>  8  1987    53.9
#>  9  1992    55.8
#> 10  1997    56.5
#> 11  2002    56.8
#> 12  2007    59.7
```

6.7 Pure, predictable, pipeable

We've barely scratched the surface of dplyr but I want to point out key principles you may start to appreciate. If you're new to R or "programming with data", feel free skip this section and move on.

dplyr's verbs, such as `filter()` and `select()`, are what's called pure functions. To quote from the Functions chapter of Wickham's Advanced R book (2015):

> The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no side effects: they don't affect the state of the world in any way apart from the value they return.

In fact, these verbs are a special case of pure functions: they take the same flavor of object as input and output. Namely, a data frame or one of the other data receptacles dplyr supports.

And finally, the data is **always** the very first argument of the verb functions.

This set of deliberate design choices, together with the new pipe operator, produces a highly effective, low friction domain-specific language for data analysis.

Go to the next Chapter, dplyr functions for a single dataset, for more dplyr!

6.8 Resources

dplyr official stuff:

- Package home on CRAN.
  - Note there are several vignettes, with the Introduction to dplyr being the most relevant right now.
  - The Window functions one will also be interesting to you now.
- Development home on GitHub.
- Tutorial HW delivered (note this links to a DropBox folder) at useR! 2014 conference.

RStudio Data Transformation Cheat Sheet, covering dplyr. Remember you can get to these via *Help > Cheatsheets*.

Data transformation chapter of R for Data Science (Wickham and Grolemund, 2016).

"Let the Data Flow: Pipelines in R with dplyr and magrittr" - Excellent slides on pipelines and dplyr by TJ Mahr, talk given to the Madison R Users Group.

Blog post "Hands-on dplyr tutorial for faster data manipulation in R" by Data School, that includes a link to an R Markdown document and links to videos.

Chapter 15 - cheatsheet I made for dplyr join functions (not relevant yet but soon).

Chapter 7

Single table dplyr functions

7.1 Where were we?

In Chapter 6, Introduction to dplyr, we used two very important verbs and an operator:

- `filter()` for subsetting data with row logic
- `select()` for subsetting data variable- or column-wise
- the pipe operator `%>%`, which feeds the LHS as the first argument to the expression on the RHS

We also discussed dplyr's role inside the tidyverse and tibbles:

- dplyr is a core package in the tidyverse meta-package. Since we often make incidental usage of the others, we will load dplyr and the others via `library(tidyverse)`.
- The tidyverse embraces a special flavor of data frame, called a tibble. The `gapminder` dataset is stored as a tibble.

7.2 Load dplyr and gapminder

I choose to load the tidyverse, which will load dplyr, among other packages we use incidentally below.

```
library(tidyverse)
#>  Attaching packages                    tidyverse 1.3.0
#>  ggplot2 3.3.2       purrr   0.3.4
```

```
#>   tibble  3.0.3      dplyr   1.0.2
#>   tidyr   1.1.2      stringr 1.4.0
#>   readr   1.3.1      forcats 0.5.0
#>    Conflicts                  tidyverse_conflicts()
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Also load gapminder.

```
library(gapminder)
```

7.3 Create a copy of `gapminder`

We're going to make changes to the `gapminder` tibble. To eliminate any fear that you're damaging the data that comes with the package, we create an explicit copy of `gapminder` for our experiments.

```
(my_gap <- gapminder)
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # … with 1,694 more rows
```

Pay close attention to when we evaluate statements but let the output just print to screen:

```
## let output print to screen, but do not store
my_gap %>% filter(country == "Canada")
```

… versus when we assign the output to an object, possibly overwriting an existing object.

```
## store the output as an R object
my_precious <- my_gap %>% filter(country == "Canada")
```

7.4 Use `mutate()` to add new variables

Imagine we wanted to recover each country's GDP. After all, the Gapminder data has a variable for population and GDP per capita. Let's multiply them together.

`mutate()` is a function that defines and inserts new variables into a tibble. You can refer to existing variables by name.

```
my_gap %>%
  mutate(gdp = pop * gdpPercap)
#> # A tibble: 1,704 x 7
#>    country     continent  year lifeExp      pop gdpPercap          gdp
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.      6.57e 9
#>  2 Afghanistan Asia       1957    30.3  9240934      821.      7.59e 9
#>  3 Afghanistan Asia       1962    32.0 10267083      853.      8.76e 9
#>  4 Afghanistan Asia       1967    34.0 11537966      836.      9.65e 9
#>  5 Afghanistan Asia       1972    36.1 13079460      740.      9.68e 9
#>  6 Afghanistan Asia       1977    38.4 14880372      786.      1.17e10
#>  7 Afghanistan Asia       1982    39.9 12881816      978.      1.26e10
#>  8 Afghanistan Asia       1987    40.8 13867957      852.      1.18e10
#>  9 Afghanistan Asia       1992    41.7 16317921      649.      1.06e10
#> 10 Afghanistan Asia       1997    41.8 22227415      635.      1.41e10
#> # … with 1,694 more rows
```

Hmmmm … those GDP numbers are almost uselessly large and abstract. Consider the advice of Randall Munroe of xkcd:

> One thing that bothers me is large numbers presented without context… "If I added a zero to this number, would the sentence containing it mean something different to me?" If the answer is "no", maybe the number has no business being in the sentence in the first place.

Maybe it would be more meaningful to consumers of my tables and figures to stick with GDP per capita. But what if I reported GDP per capita, *relative to some benchmark country*. Since Canada is my adopted home, I'll go with that.

I need to create a new variable that is `gdpPercap` divided by Canadian `gdpPercap`, taking care that I always divide two numbers that pertain to the same year.

How I achieve this:

1. Filter down to the rows for Canada.
2. Create a new temporary variable in `my_gap`:
   i) Extract the `gdpPercap` variable from the Canadian data.
   ii) Replicate it once per country in the dataset, so it has the right length.
3. Divide raw `gdpPercap` by this Canadian figure.
4. Discard the temporary variable of replicated Canadian `gdpPercap`.

```
ctib <- my_gap %>%
  filter(country == "Canada")
## this is a semi-dangerous way to add this variable
## I'd prefer to join on year, but we haven't covered joins yet
my_gap <- my_gap %>%
  mutate(tmp = rep(ctib$gdpPercap, nlevels(country)),
         gdpPercapRel = gdpPercap / tmp,
         tmp = NULL)
```

Note that, `mutate()` builds new variables sequentially so you can reference earlier ones (like `tmp`) when defining later ones (like `gdpPercapRel`). Also, you can get rid of a variable by setting it to `NULL`.

How could we sanity check that this worked? The Canadian values for `gdpPercapRel` better all be 1!

```
my_gap %>%
  filter(country == "Canada") %>%
  select(country, year, gdpPercapRel)
#> # A tibble: 12 x 3
#>    country  year gdpPercapRel
#>    <fct>   <int>        <dbl>
#>  1 Canada   1952            1
#>  2 Canada   1957            1
#>  3 Canada   1962            1
#>  4 Canada   1967            1
#>  5 Canada   1972            1
#>  6 Canada   1977            1
#>  7 Canada   1982            1
#>  8 Canada   1987            1
#>  9 Canada   1992            1
#> 10 Canada   1997            1
#> 11 Canada   2002            1
#> 12 Canada   2007            1
```

I perceive Canada to be a "high GDP" country, so I predict that the distribution of `gdpPercapRel` is located below 1, possibly even well below. Check your intuition!

```
summary(my_gap$gdpPercapRel)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    0.01    0.06    0.17    0.33    0.45    9.53
```

The relative GDP per capita numbers are, in general, well below 1. We see that most of the countries covered by this dataset have substantially lower GDP per capita, relative to Canada, across the entire time period.

Remember: Trust No One. Including (especially?) yourself. Always try to find a way to check that you've done what meant to. Prepare to be horrified.

7.5 Use `arrange()` to row-order data in a principled way

`arrange()` reorders the rows in a data frame. Imagine you wanted this data ordered by year then country, as opposed to by country then year.

```
my_gap %>%
  arrange(year, country)
#> # A tibble: 1,704 x 7
#>    country     continent  year lifeExp     pop gdpPercap gdpPercapRel
#>    <fct>       <fct>     <int>   <dbl>   <int>     <dbl>        <dbl>
#>  1 Afghanistan Asia       1952    28.8  8.43e6      779.       0.0686
#>  2 Albania     Europe     1952    55.2  1.28e6     1601.       0.141
#>  3 Algeria     Africa     1952    43.1  9.28e6     2449.       0.215
#>  4 Angola      Africa     1952    30.0  4.23e6     3521.       0.310
#>  5 Argentina   Americas   1952    62.5  1.79e7     5911.       0.520
#>  6 Australia   Oceania    1952    69.1  8.69e6    10040.       0.883
#>  7 Austria     Europe     1952    66.8  6.93e6     6137.       0.540
#>  8 Bahrain     Asia       1952    50.9  1.20e5     9867.       0.868
#>  9 Bangladesh  Asia       1952    37.5  4.69e7      684.       0.0602
#> 10 Belgium     Europe     1952    68    8.73e6     8343.       0.734
#> # … with 1,694 more rows
```

Or maybe you want just the data from 2007, sorted on life expectancy?

```
my_gap %>%
  filter(year == 2007) %>%
  arrange(lifeExp)
```

```
#> # A tibble: 142 x 7
#>    country      continent  year lifeExp    pop gdpPercap gdpPercapRel
#>    <fct>        <fct>     <int>   <dbl>  <int>     <dbl>        <dbl>
#>  1 Swaziland    Africa     2007    39.6 1.13e6     4513.       0.124
#>  2 Mozambique   Africa     2007    42.1 2.00e7      824.       0.0227
#>  3 Zambia       Africa     2007    42.4 1.17e7     1271.       0.0350
#>  4 Sierra Leone Africa     2007    42.6 6.14e6      863.       0.0237
#>  5 Lesotho      Africa     2007    42.6 2.01e6     1569.       0.0432
#>  6 Angola       Africa     2007    42.7 1.24e7     4797.       0.132
#>  7 Zimbabwe     Africa     2007    43.5 1.23e7      470.       0.0129
#>  8 Afghanistan  Asia       2007    43.8 3.19e7      975.       0.0268
#>  9 Central Afr… Africa     2007    44.7 4.37e6      706.       0.0194
#> 10 Liberia      Africa     2007    45.7 3.19e6      415.       0.0114
#> # … with 132 more rows
```

Oh, you'd like to sort on life expectancy in **desc**ending order? Then use `desc()`.

```
my_gap %>%
  filter(year == 2007) %>%
  arrange(desc(lifeExp))
#> # A tibble: 142 x 7
#>    country     continent  year lifeExp     pop gdpPercap gdpPercapRel
#>    <fct>       <fct>     <int>   <dbl>   <int>     <dbl>        <dbl>
#>  1 Japan       Asia       2007    82.6  1.27e8    31656.        0.872
#>  2 Hong Kong,… Asia       2007    82.2  6.98e6    39725.        1.09
#>  3 Iceland     Europe     2007    81.8  3.02e5    36181.        0.996
#>  4 Switzerland Europe     2007    81.7  7.55e6    37506.        1.03
#>  5 Australia   Oceania    2007    81.2  2.04e7    34435.        0.948
#>  6 Spain       Europe     2007    80.9  4.04e7    28821.        0.794
#>  7 Sweden      Europe     2007    80.9  9.03e6    33860.        0.932
#>  8 Israel      Asia       2007    80.7  6.43e6    25523.        0.703
#>  9 France      Europe     2007    80.7  6.11e7    30470.        0.839
#> 10 Canada      Americas   2007    80.7  3.34e7    36319.        1
#> # … with 132 more rows
```

I advise that your analyses NEVER rely on rows or variables being in a specific order. But it's still true that human beings write the code and the interactive development process can be much nicer if you reorder the rows of your data as you go along. Also, once you are preparing tables for human eyeballs, it is imperative that you step up and take control of row order.

7.6 Use `rename()` to rename variables

When I first cleaned this Gapminder excerpt, I was a `camelCase` person, but now I'm all about `snake_case`. So I am vexed by the variable names I chose when I cleaned this data years ago. Let's rename some variables!

```
my_gap %>%
  rename(life_exp = lifeExp,
         gdp_percap = gdpPercap,
         gdp_percap_rel = gdpPercapRel)
#> # A tibble: 1,704 x 7
#>    country  continent  year life_exp    pop gdp_percap gdp_percap_rel
#>    <fct>    <fct>     <int>    <dbl>  <int>      <dbl>          <dbl>
#>  1 Afghani… Asia       1952     28.8 8.43e6       779.         0.0686
#>  2 Afghani… Asia       1957     30.3 9.24e6       821.         0.0657
#>  3 Afghani… Asia       1962     32.0 1.03e7       853.         0.0634
#>  4 Afghani… Asia       1967     34.0 1.15e7       836.         0.0520
#>  5 Afghani… Asia       1972     36.1 1.31e7       740.         0.0390
#>  6 Afghani… Asia       1977     38.4 1.49e7       786.         0.0356
#>  7 Afghani… Asia       1982     39.9 1.29e7       978.         0.0427
#>  8 Afghani… Asia       1987     40.8 1.39e7       852.         0.0320
#>  9 Afghani… Asia       1992     41.7 1.63e7       649.         0.0246
#> 10 Afghani… Asia       1997     41.8 2.22e7       635.         0.0219
#> # … with 1,694 more rows
```

I did NOT assign the post-rename object back to `my_gap` because that would make the chunks in this tutorial harder to copy/paste and run out of order. In real life, I would probably assign this back to `my_gap`, in a data preparation script, and proceed with the new variable names.

7.7 `select()` can rename and reposition variables

You've seen simple use of `select()`. There are two tricks you might enjoy:

1. `select()` can rename the variables you request to keep.
2. `select()` can be used with `everything()` to hoist a variable up to the front of the tibble.

```
my_gap %>%
  filter(country == "Burundi", year > 1996) %>%
  select(yr = year, lifeExp, gdpPercap) %>%
```

```
  select(gdpPercap, everything())
#> # A tibble: 3 x 3
#>   gdpPercap    yr lifeExp
#>       <dbl> <int>   <dbl>
#> 1      463.  1997    45.3
#> 2      446.  2002    47.4
#> 3      430.  2007    49.6
```

`everything()` is one of several helpers for variable selection. Read its help to see the rest.

7.8 `group_by()` is a mighty weapon

I have found that ~~friends and family~~ collaborators love to ask seemingly innocuous questions like, "which country experienced the sharpest 5-year drop in life expectancy?". In fact, that is a totally natural question to ask. But if you are using a language that doesn't know about data, it's an incredibly annoying question to answer.

dplyr offers powerful tools to solve this class of problem:

- `group_by()` adds extra structure to your dataset – grouping information – which lays the groundwork for computations within the groups.
- `summarize()` takes a dataset with n observations, computes requested summaries, and returns a dataset with 1 observation.
- Window functions take a dataset with n observations and return a dataset with n observations.
- `mutate()` and `summarize()` will honor groups.
- You can also do very general computations on your groups with `do()`, though elsewhere in this course, I advocate for other approaches that I find more intuitive, using the purrr package.

Combined with the verbs you already know, these new tools allow you to solve an extremely diverse set of problems with relative ease.

7.8.1 Counting things up

Let's start with simple counting. How many observations do we have per continent?

```
my_gap %>%
  group_by(continent) %>%
  summarize(n = n())
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 5 x 2
#>   continent     n
#>   <fct>     <int>
#> 1 Africa      624
#> 2 Americas    300
#> 3 Asia        396
#> 4 Europe      360
#> 5 Oceania      24
```

Let us pause here to think about the tidyverse. You could get these same frequencies using `table()` from base R.

```
table(gapminder$continent)
#> 
#>   Africa Americas     Asia   Europe  Oceania 
#>      624      300      396      360       24
str(table(gapminder$continent))
#>  'table' int [1:5(1d)] 624 300 396 360 24
#>  - attr(*, "dimnames")=List of 1
#>   ..$ : chr [1:5] "Africa" "Americas" "Asia" "Europe" ...
```

But the object of class `table` that is returned makes downstream computation a bit fiddlier than you'd like. For example, it's too bad the continent levels come back only as *names* and not as a proper factor, with the original set of levels. This is an example of how the tidyverse smooths transitions where you want the output of step `i` to become the input of step `i + 1`.

The `tally()` function is a convenience function that knows to count rows. It honors groups.

```
my_gap %>%
  group_by(continent) %>%
  tally()
#> # A tibble: 5 x 2
#>   continent     n
#>   <fct>     <int>
#> 1 Africa      624
#> 2 Americas    300
#> 3 Asia        396
#> 4 Europe      360
#> 5 Oceania      24
```

The `count()` function is an even more convenient function that does both grouping and counting.

```
my_gap %>%
  count(continent)
#> # A tibble: 5 x 2
#>   continent     n
#>   <fct>     <int>
#> 1 Africa      624
#> 2 Americas    300
#> 3 Asia        396
#> 4 Europe      360
#> 5 Oceania      24
```

What if we wanted to add the number of unique countries for each continent? You can compute multiple summaries inside `summarize()`. Use the `n_distinct()` function to count the number of distinct countries within each continent.

```
my_gap %>%
  group_by(continent) %>%
  summarize(n = n(),
            n_countries = n_distinct(country))
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 5 x 3
#>   continent     n n_countries
#>   <fct>     <int>       <int>
#> 1 Africa      624          52
#> 2 Americas    300          25
#> 3 Asia        396          33
#> 4 Europe      360          30
#> 5 Oceania      24           2
```

7.8.2 General summarization

The functions you'll apply within `summarize()` include classical statistical summaries, like `mean()`, `median()`, `var()`, `sd()`, `mad()`, `IQR()`, `min()`, and `max()`. Remember they are functions that take n inputs and distill them down into 1 output.

Although this may be statistically ill-advised, let's compute the average life expectancy by continent.

```
my_gap %>%
  group_by(continent) %>%
  summarize(avg_lifeExp = mean(lifeExp))
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 5 x 2
#>   continent avg_lifeExp
#>   <fct>           <dbl>
#> 1 Africa           48.9
#> 2 Americas         64.7
#> 3 Asia             60.1
#> 4 Europe           71.9
#> 5 Oceania          74.3
```

`summarize_at()` applies the same summary function(s) to multiple variables. Let's compute average and median life expectancy and GDP per capita by continent by year…but only for 1952 and 2007.

```
my_gap %>%
  filter(year %in% c(1952, 2007)) %>%
  group_by(continent, year) %>%
  summarize_at(vars(lifeExp, gdpPercap), list(~mean(.), ~median(.)))
#> # A tibble: 10 x 6
#> # Groups:   continent [5]
#>    continent  year lifeExp_mean gdpPercap_mean lifeExp_median
#>    <fct>     <int>        <dbl>          <dbl>          <dbl>
#>  1 Africa     1952         39.1          1253.           38.8
#>  2 Africa     2007         54.8          3089.           52.9
#>  3 Americas   1952         53.3          4079.           54.7
#>  4 Americas   2007         73.6         11003.           72.9
#>  5 Asia       1952         46.3          5195.           44.9
#>  6 Asia       2007         70.7         12473.           72.4
#>  7 Europe     1952         64.4          5661.           65.9
#>  8 Europe     2007         77.6         25054.           78.6
#>  9 Oceania    1952         69.3         10298.           69.3
#> 10 Oceania    2007         80.7         29810.           80.7
#> # … with 1 more variable: gdpPercap_median <dbl>
```

Let's focus just on Asia. What are the minimum and maximum life expectancies seen by year?

```
my_gap %>%
  filter(continent == "Asia") %>%
  group_by(year) %>%
  summarize(min_lifeExp = min(lifeExp), max_lifeExp = max(lifeExp))
```

```
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 12 x 3
#>     year min_lifeExp max_lifeExp
#>    <int>       <dbl>       <dbl>
#>  1  1952        28.8        65.4
#>  2  1957        30.3        67.8
#>  3  1962        32.0        69.4
#>  4  1967        34.0        71.4
#>  5  1972        36.1        73.4
#>  6  1977        31.2        75.4
#>  7  1982        39.9        77.1
#>  8  1987        40.8        78.7
#>  9  1992        41.7        79.4
#> 10  1997        41.8        80.7
#> 11  2002        42.1        82
#> 12  2007        43.8        82.6
```

Of course it would be much more interesting to see *which* country contributed these extreme observations. Is the minimum (maximum) always coming from the same country? We tackle that with window functions shortly.

7.9 Grouped mutate

Sometimes you don't want to collapse the n rows for each group into one row. You want to keep your groups, but compute within them.

7.9.1 Computing with group-wise summaries

Let's make a new variable that is the years of life expectancy gained (lost) relative to 1952, for each individual country. We group by country and use `mutate()` to make a new variable. The `first()` function extracts the first value from a vector. Notice that `first()` is operating on the vector of life expectancies *within each country group*.

```
my_gap %>%
  group_by(country) %>%
  select(country, year, lifeExp) %>%
  mutate(lifeExp_gain = lifeExp - first(lifeExp)) %>%
  filter(year < 1963)
#> # A tibble: 426 x 4
#> # Groups:   country [142]
#>    country      year lifeExp lifeExp_gain
```

```
#>    <fct>        <int>   <dbl>        <dbl>
#>  1 Afghanistan  1952    28.8         0
#>  2 Afghanistan  1957    30.3         1.53
#>  3 Afghanistan  1962    32.0         3.20
#>  4 Albania      1952    55.2         0
#>  5 Albania      1957    59.3         4.05
#>  6 Albania      1962    64.8         9.59
#>  7 Algeria      1952    43.1         0
#>  8 Algeria      1957    45.7         2.61
#>  9 Algeria      1962    48.3         5.23
#> 10 Angola       1952    30.0         0
#> # … with 416 more rows
```

Within country, we take the difference between life expectancy in year i and life expectancy in 1952. Therefore we always see zeroes for 1952 and, for most countries, a sequence of positive and increasing numbers.

7.9.2 Window functions

Window functions take n inputs and give back n outputs. Furthermore, the output depends on all the values. So `rank()` is a window function but `log()` is not. Here we use window functions based on ranks and offsets.

Let's revisit the worst and best life expectancies in Asia over time, but retaining info about *which* country contributes these extreme values.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year) %>%
  filter(min_rank(desc(lifeExp)) < 2 | min_rank(lifeExp) < 2) %>%
  arrange(year) %>%
  print(n = Inf)
#> # A tibble: 24 x 3
#> # Groups:   year [12]
#>     year country     lifeExp
#>    <int> <fct>         <dbl>
#>  1  1952 Afghanistan    28.8
#>  2  1952 Israel         65.4
#>  3  1957 Afghanistan    30.3
#>  4  1957 Israel         67.8
#>  5  1962 Afghanistan    32.0
#>  6  1962 Israel         69.4
#>  7  1967 Afghanistan    34.0
```

```
#>  8  1967 Japan          71.4
#>  9  1972 Afghanistan    36.1
#> 10  1972 Japan          73.4
#> 11  1977 Cambodia       31.2
#> 12  1977 Japan          75.4
#> 13  1982 Afghanistan    39.9
#> 14  1982 Japan          77.1
#> 15  1987 Afghanistan    40.8
#> 16  1987 Japan          78.7
#> 17  1992 Afghanistan    41.7
#> 18  1992 Japan          79.4
#> 19  1997 Afghanistan    41.8
#> 20  1997 Japan          80.7
#> 21  2002 Afghanistan    42.1
#> 22  2002 Japan          82
#> 23  2007 Afghanistan    43.8
#> 24  2007 Japan          82.6
```

We see that (min = Afghanistan, max = Japan) is the most frequent result, but Cambodia and Israel pop up at least once each as the min or max, respectively. That table should make you impatient for our upcoming work on tidying and reshaping data! Wouldn't it be nice to have one row per year?

How did that actually work? First, I store and view a partial that leaves off the `filter()` statement. All of these operations should be familiar.

```
asia <- my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year)
asia
#> # A tibble: 396 x 3
#> # Groups:   year [12]
#>     year country     lifeExp
#>    <int> <fct>         <dbl>
#>  1  1952 Afghanistan    28.8
#>  2  1957 Afghanistan    30.3
#>  3  1962 Afghanistan    32.0
#>  4  1967 Afghanistan    34.0
#>  5  1972 Afghanistan    36.1
#>  6  1977 Afghanistan    38.4
#>  7  1982 Afghanistan    39.9
#>  8  1987 Afghanistan    40.8
#>  9  1992 Afghanistan    41.7
#> 10  1997 Afghanistan    41.8
#> # … with 386 more rows
```

Now we apply a window function – `min_rank()`. Since `asia` is grouped by year, `min_rank()` operates within mini-datasets, each for a specific year. Applied to the variable `lifeExp`, `min_rank()` returns the rank of each country's observed life expectancy. FYI, the `min` part just specifies how ties are broken. Here is an explicit peek at these within-year life expectancy ranks, in both the (default) ascending and descending order.

For concreteness, I use `mutate()` to actually create these variables, even though I dropped this in the solution above. Let's look at a bit of that.

```
asia %>%
  mutate(le_rank = min_rank(lifeExp),
         le_desc_rank = min_rank(desc(lifeExp))) %>%
  filter(country %in% c("Afghanistan", "Japan", "Thailand"), year > 1995)
#> # A tibble: 9 x 5
#> # Groups:   year [3]
#>    year country     lifeExp le_rank le_desc_rank
#>   <int> <fct>         <dbl>   <int>        <int>
#> 1  1997 Afghanistan    41.8       1           33
#> 2  2002 Afghanistan    42.1       1           33
#> 3  2007 Afghanistan    43.8       1           33
#> 4  1997 Japan          80.7      33            1
#> 5  2002 Japan          82        33            1
#> 6  2007 Japan          82.6      33            1
#> 7  1997 Thailand       67.5      12           22
#> 8  2002 Thailand       68.6      12           22
#> 9  2007 Thailand       70.6      12           22
```

Afghanistan tends to present 1's in the `le_rank` variable, Japan tends to present 1's in the `le_desc_rank` variable and other countries, like Thailand, present less extreme ranks.

You can understand the original `filter()` statement now:

```
filter(min_rank(desc(lifeExp)) < 2 | min_rank(lifeExp) < 2)
```

These two sets of ranks are formed on-the-fly, within year group, and `filter()` retains rows with rank less than 2, which means … the row with rank = 1. Since we do for ascending and descending ranks, we get both the min and the max.

If we had wanted just the min OR the max, an alternative approach using `top_n()` would have worked.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
```

```
  arrange(year) %>%
  group_by(year) %>%
  #top_n(1, wt = lifeExp)        ## gets the min
  top_n(1, wt = desc(lifeExp)) ## gets the max
#> # A tibble: 12 x 3
#> # Groups:   year [12]
#>     year country     lifeExp
#>    <int> <fct>         <dbl>
#>  1  1952 Afghanistan    28.8
#>  2  1957 Afghanistan    30.3
#>  3  1962 Afghanistan    32.0
#>  4  1967 Afghanistan    34.0
#>  5  1972 Afghanistan    36.1
#>  6  1977 Cambodia       31.2
#>  7  1982 Afghanistan    39.9
#>  8  1987 Afghanistan    40.8
#>  9  1992 Afghanistan    41.7
#> 10  1997 Afghanistan    41.8
#> 11  2002 Afghanistan    42.1
#> 12  2007 Afghanistan    43.8
```

7.10 Grand Finale

So let's answer that "simple" question: which country experienced the sharpest 5-year drop in life expectancy? Recall that this excerpt of the Gapminder data only has data every five years, e.g. for 1952, 1957, etc. So this really means looking at life expectancy changes between adjacent timepoints.

At this point, that's just too easy, so let's do it by continent while we're at it.

```
my_gap %>%
  select(country, year, continent, lifeExp) %>%
  group_by(continent, country) %>%
  ## within country, take (lifeExp in year i) - (lifeExp in year i - 1)
  ## positive means lifeExp went up, negative means it went down
  mutate(le_delta = lifeExp - lag(lifeExp)) %>%
  ## within country, retain the worst lifeExp change = smallest or most negative
  summarize(worst_le_delta = min(le_delta, na.rm = TRUE)) %>%
  ## within continent, retain the row with the lowest worst_le_delta
  top_n(-1, wt = worst_le_delta) %>%
  arrange(worst_le_delta)
#> `summarise()` regrouping output by 'continent' (override with `.groups` argument)
#> # A tibble: 5 x 3
#> # Groups:   continent [5]
```

```
#>   continent country     worst_le_delta
#>   <fct>     <fct>                <dbl>
#> 1 Africa    Rwanda             -20.4
#> 2 Asia      Cambodia            -9.10
#> 3 Americas  El Salvador         -1.51
#> 4 Europe    Montenegro          -1.46
#> 5 Oceania   Australia            0.170
```

Ponder that for a while. The subject matter and the code. Mostly you're seeing what genocide looks like in dry statistics on average life expectancy.

Break the code into pieces, starting at the top, and inspect the intermediate results. That's certainly how I was able to *write* such a thing. These commands do not leap fully formed out of anyone's forehead – they are built up gradually, with lots of errors and refinements along the way. I'm not even sure it's a great idea to do so much manipulation in one fell swoop. Is the statement above really hard for you to read? If yes, then by all means break it into pieces and make some intermediate objects. Your code should be easy to write and read when you're done.

In later tutorials, we'll explore more of dplyr, such as operations based on two datasets.

7.11 Resources

dplyr official stuff:

- Package home on CRAN.
  - Note there are several vignettes, with the Introduction to dplyr being the most relevant right now.
  - The Window functions one will also be interesting to you now.
- Development home on GitHub.
- Tutorial HW delivered (note this links to a DropBox folder) at useR! 2014 conference.

RStudio Data Transformation Cheat Sheet, covering dplyr. Remember you can get to these via *Help > Cheatsheets.*

Data transformation chapter of R for Data Science (Wickham and Grolemund, 2016).

"Let the Data Flow: Pipelines in R with dplyr and magrittr" - Excellent slides on pipelines and dplyr by TJ Mahr, talk given to the Madison R Users Group.

Blog post "Hands-on dplyr tutorial for faster data manipulation in R" by Data School, that includes a link to an R Markdown document and links to videos.

Chapter 15 - cheatsheet I made for dplyr join functions (not relevant yet but soon).

Chapter 8

Tidy data

Tidy data using Lord of the Rings: tidy data, tidyr.

Chapter 9

Writing and reading files

9.1 File I/O overview

We've been loading the Gapminder data as a data frame from the gapminder data package. We haven't been explicitly writing any data or derived results to file. In real life, you'll bring rectangular data into and out of R all the time. Sometimes you'll need to do same for non-rectangular objects.

How do you do this? What issues should you think about?

9.1.1 Data import mindset

Data import generally feels one of two ways:

- *"Surprise me!"* This is the attitude you must adopt when you first get a dataset. You are just happy to import without an error. You start to explore. You discover flaws in the data and/or the import. You address them. Lather, rinse, repeat.
- *"Another day in paradise."* This is the attitude when you bring in a tidy dataset you have maniacally cleaned in one or more cleaning scripts. There should be no surprises. You should express your expectations about the data in formal assertions at the very start of these downstream scripts.

In the second case, and as the first cases progresses, you actually know a lot about how the data is/should be. My main import advice: **use the arguments of your import function to get as far as you can, as fast as possible**. Novice code often has a great deal of unnecessary post import fussing around. Read the docs for the import functions and take maximum advantage of the arguments to control the import.

9.1.2 Data export mindset

There will be many occasions when you need to write data from R. Two main examples:

- a tidy ready-to-analyze dataset that you heroically created from messy data
- a numerical result from data aggregation or modelling or statistical inference

First tip: **today's outputs are tomorrow's inputs**. Think back on all the pain you have suffered importing data and don't inflict such pain on yourself!

Second tip: don't be too cute or clever. A plain text file that is readable by a human being in a text editor should be your default until you have **actual proof** that this will not work. Reading and writing to exotic or proprietary formats will be the first thing to break in the future or on a different computer. It also creates barriers for anyone who has a different toolkit than you do. Be software-agnostic. Aim for future-proof and moron-proof.

How does this fit with our emphasis on dynamic reporting via R Markdown? There is a time and place for everything. There are projects and documents where the scope and personnel will allow you to geek out with knitr and R Markdown. But there are lots of good reasons why (parts of) an analysis should not (only) be embedded in a dynamic report. Maybe you are just doing data cleaning to produce a valid input dataset. Maybe you are making a small but crucial contribution to a giant multi-author paper. Etc. Also remember there are other tools and workflows for making something reproducible. I'm looking at you, make.

9.2 Load the tidyverse

The main package we will be using is readr, which provides drop-in substitute functions for `read.table()` and friends. However, to make some points about data export and import, it is nice to reorder factor levels. For that, we will use the forcats package, which is also included in the tidyverse meta-package.

```
library(tidyverse)
#>  Attaching packages                  tidyverse 1.3.0
#>  ggplot2 3.3.2      purrr   0.3.4
#>  tibble  3.0.3      dplyr   1.0.2
#>  tidyr   1.1.2      stringr 1.4.0
#>  readr   1.3.1      forcats 0.5.0
#>  Conflicts                   tidyverse_conflicts()
```

```
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

9.3 Locate the Gapminder data

We could load the data from the package as usual, but instead we will load it from tab delimited file. The gapminder package includes the data normally found in the `gapminder` data frame as a `.tsv`. So let's get the path to that file on *your* system using the fs package.

```
library(fs)
(gap_tsv <- path_package("gapminder", "extdata", "gapminder.tsv"))
#> /Users/hgstp/Library/R/4.0/library/gapminder/extdata/gapminder.tsv
```

9.4 Bring rectangular data in

The workhorse data import function of readr is `read_delim()`. Here we'll use a variant, `read_tsv()`, that anticipates tab-delimited data:

```
gapminder <- read_tsv(gap_tsv)
#> Parsed with column specification:
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   year = col_double(),
#>   lifeExp = col_double(),
#>   pop = col_double(),
#>   gdpPercap = col_double()
#> )
str(gapminder, give.attr = FALSE)
#> tibble [1,704 × 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
#>  $ country  : chr [1:1704] "Afghanistan" "Afghanistan" "Afghanista"..
#>  $ continent: chr [1:1704] "Asia" "Asia" "Asia" "Asia" ...
#>  $ year     : num [1:1704] 1952 1957 1962 1967 1972 ...
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
#>  $ pop      : num [1:1704] 8425333 9240934 10267083 11537966 130794..
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

For full flexibility re: specifying the delimiter, you can always use `readr::read_delim()`.

There's a similar convenience wrapper for comma-separated values: `read_csv()`.

The most noticeable difference between the readr functions and base is that readr does NOT convert strings to factors by default. In the grand scheme of things, this is better default behavior, although we go ahead and convert them to factor here. Do not be deceived – in general, you will do less post-import fussing if you use readr.

```
gapminder <- gapminder %>%
  mutate(country = factor(country),
         continent = factor(continent))
str(gapminder)
#> tibble [1,704 × 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 ..
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 ..
#>  $ year     : num [1:1704] 1952 1957 1962 1967 1972 ...
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
#>  $ pop      : num [1:1704] 8425333 9240934 10267083 11537966 130794..
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
#>  - attr(*, "spec")=
#>   .. cols(
#>   ..   country = col_character(),
#>   ..   continent = col_character(),
#>   ..   year = col_double(),
#>   ..   lifeExp = col_double(),
#>   ..   pop = col_double(),
#>   ..   gdpPercap = col_double()
#>   .. )
```

9.4.1 Bring rectangular data in – summary

Default to `readr::read_delim()` and friends. Use the arguments!

The Gapminder data is too clean and simple to show off the great features of readr, so I encourage you to check out the part of the introduction vignette on column types. There are many variable types that you will be able to parse correctly upon import, thereby eliminating a great deal of post-import fussing.

9.5 Compute something worthy of export

We need compute something worth writing to file. Let's create a country-level summary of maximum life expectancy.

```
gap_life_exp <- gapminder %>%
  group_by(country, continent) %>%
  summarise(life_exp = max(lifeExp)) %>%
  ungroup()
#> `summarise()` regrouping output by 'country' (override with `.groups` argument)
gap_life_exp
#> # A tibble: 142 x 3
#>    country     continent life_exp
#>    <fct>       <fct>        <dbl>
#>  1 Afghanistan Asia          43.8
#>  2 Albania     Europe        76.4
#>  3 Algeria     Africa        72.3
#>  4 Angola      Africa        42.7
#>  5 Argentina   Americas      75.3
#>  6 Australia   Oceania       81.2
#>  7 Austria     Europe        79.8
#>  8 Bahrain     Asia          75.6
#>  9 Bangladesh  Asia          64.1
#> 10 Belgium     Europe        79.4
#> # … with 132 more rows
```

The `gap_life_exp` data frame is an example of an intermediate result that we want to store for the future and for downstream analyses or visualizations.

9.6 Write rectangular data out

The workhorse export function for rectangular data in readr is `write_delim()` and friends. Let's use `write_csv()` to get a comma-delimited file.

```
write_csv(gap_life_exp, "gap_life_exp.csv")
```

Let's look at the first few lines of `gap_life_exp.csv`. If you're following along, you should be able to open this file or, in a shell, use `head()` on it.

```
country,continent,life_exp
Afghanistan,Asia,43.828
Albania,Europe,76.423
Algeria,Africa,72.301
Angola,Africa,42.731
Argentina,Americas,75.32
```

This is pretty decent looking, though there is no visible alignment or separation into columns. Had we used the base function `read.csv()`, we would be seeing rownames and lots of quotes, unless we had explicitly shut that down. Nicer default behavior is the main reason we are using `readr::write_csv()` over `write.csv()`.

- It's not really fair to complain about the lack of visible alignment. Remember we are "writing data for computers". If you really want to browse around the file, use `View()` in RStudio or open it in Microsoft Excel (!) but don't succumb to the temptation to start doing artisanal data manipulations there … get back to R and construct commands that you can re-run the next 15 times you import/clean/aggregate/export the same dataset. Trust me, it will happen.

9.7 Invertibility

It turns out these self-imposed rules are often in conflict with one another:

- Write to plain text files
- Break analysis into pieces: the output of script `i` is an input for script `i + 1`
- Be the boss of factors: order the levels in a meaningful, usually non-alphabetical way
- Avoid duplication of code and data

Example: after performing the country-level summarization, we reorder the levels of the country factor, based on life expectancy. This reordering operation is conceptually important and must be embodied in R commands stored in a script. However, as soon as we write `gap_life_exp` to a plain text file, that meta-information about the countries is lost. Upon re-import with `read_delim()` and friends, we are back to alphabetically ordered factor levels. Any measure we take to avoid this loss immediately breaks another one of our rules.

So what do I do? I must admit I save (and re-load) R-specific binary files. Right after I save the plain text file. Belt and suspenders.

I have toyed with the idea of writing import helper functions for a specific project, that would re-order factor levels in principled ways. They could be defined in one file and called from many. This would also have a very natural implementation within a workflow where each analytical project is an R package. But so far it has seemed too much like yak shaving. I'm intrigued by a recent discussion of putting such information in YAML frontmatter (see Martin Fenner blog post, "Using YAML frontmatter with CSV").

9.8 Reordering the levels of the country factor

The topic of factor level reordering is covered in Chapter 10, so let's Just. Do. It. I reorder the country factor levels according to the life expectancy summary we've already computed.

```
head(levels(gap_life_exp$country)) # alphabetical order
#> [1] "Afghanistan" "Albania"     "Algeria"     "Angola"
#> [5] "Argentina"   "Australia"
gap_life_exp <- gap_life_exp %>%
  mutate(country = fct_reorder(country, life_exp))
head(levels(gap_life_exp$country)) # in increasing order of maximum life expectancy
#> [1] "Sierra Leone" "Angola"       "Afghanistan"  "Liberia"
#> [5] "Rwanda"       "Mozambique"
head(gap_life_exp)
#> # A tibble: 6 x 3
#>   country     continent life_exp
#>   <fct>       <fct>        <dbl>
#> 1 Afghanistan Asia          43.8
#> 2 Albania     Europe        76.4
#> 3 Algeria     Africa        72.3
#> 4 Angola      Africa        42.7
#> 5 Argentina   Americas      75.3
#> 6 Australia   Oceania       81.2
```

Note that the **row order of `gap_life_exp` has not changed**. I could choose to reorder the rows of the data frame if, for example, I was about to prepare a table to present to people. But I'm not, so I won't.

9.9 `saveRDS()` and `readRDS()`

If you have a data frame AND you have exerted yourself to rationalize the factor levels, you have my blessing to save it to file in a way that will preserve this hard work upon re-import. Use `saveRDS()`.

```
saveRDS(gap_life_exp, "gap_life_exp.rds")
```

`saveRDS()` serializes an R object to a binary file. It's not a file you will able to open in an editor, diff nicely with Git(Hub), or share with non-R friends. It's a special purpose, limited use function that I use in specific situations.

The opposite of `saveRDS()` is `readRDS()`. You must assign the return value to an object. I highly recommend you assign back to the same name as before. Why confuse yourself?!?

```
rm(gap_life_exp)
gap_life_exp
#> Error in eval(expr, envir, enclos): object 'gap_life_exp' not found
gap_life_exp <- readRDS("gap_life_exp.rds")
gap_life_exp
#> # A tibble: 142 x 3
#>    country     continent life_exp
#>    <fct>       <fct>        <dbl>
#>  1 Afghanistan Asia          43.8
#>  2 Albania     Europe        76.4
#>  3 Algeria     Africa        72.3
#>  4 Angola      Africa        42.7
#>  5 Argentina   Americas      75.3
#>  6 Australia   Oceania       81.2
#>  7 Austria     Europe        79.8
#>  8 Bahrain     Asia          75.6
#>  9 Bangladesh  Asia          64.1
#> 10 Belgium     Europe        79.4
#> # … with 132 more rows
```

`saveRDS()` has more arguments, in particular `compress` for controlling compression, so read the help for more advanced usage. It is also very handy for saving non-rectangular objects, like a fitted regression model, that took a nontrivial amount of time to compute.

You will eventually hear about `save()` + `load()` and even `save.image()`. You may even see them in documentation and tutorials, but don't be tempted. Just say no. These functions encourage unsafe practices, like storing multiple objects together and even entire workspaces. There are legitimate uses of these functions, but not in your typical data analysis.

9.10 Retaining factor levels upon re-import

Concrete demonstration of how non-alphabetical factor level order is lost with `write_delim()` / `read_delim()` workflows but maintained with `saveRDS()` / `readRDS()`.

```
(country_levels <- tibble(original = head(levels(gap_life_exp$country))))
#> # A tibble: 6 x 1
#>   original
#>   <chr>
#> 1 Sierra Leone
#> 2 Angola
#> 3 Afghanistan
```

```
#> 4 Liberia
#> 5 Rwanda
#> 6 Mozambique
write_csv(gap_life_exp, "gap_life_exp.csv")
saveRDS(gap_life_exp, "gap_life_exp.rds")
rm(gap_life_exp)
head(gap_life_exp) # will cause error! proving gap_life_exp is really gone
#> Error in head(gap_life_exp): object 'gap_life_exp' not found
gap_via_csv <- read_csv("gap_life_exp.csv") %>%
  mutate(country = factor(country))
#> Parsed with column specification:
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   life_exp = col_double()
#> )
gap_via_rds <- readRDS("gap_life_exp.rds")
country_levels <- country_levels %>%
  mutate(via_csv = head(levels(gap_via_csv$country)),
         via_rds = head(levels(gap_via_rds$country)))
country_levels
#> # A tibble: 6 x 3
#>   original     via_csv     via_rds
#>   <chr>        <chr>       <chr>
#> 1 Sierra Leone Afghanistan Sierra Leone
#> 2 Angola       Albania     Angola
#> 3 Afghanistan  Algeria     Afghanistan
#> 4 Liberia      Angola      Liberia
#> 5 Rwanda       Argentina   Rwanda
#> 6 Mozambique   Australia   Mozambique
```

Note how the original, post-reordering country factor levels are restored using the `saveRDS()` / `readRDS()` strategy but revert to alphabetical ordering using `write_csv()` / `read_csv()`.

9.11 dput() and dget()

One last method of saving and restoring data deserves a mention: `dput()` and `dget()`. `dput()` offers this odd combination of features: it creates a plain text representation of an R object which still manages to be quite opaque. If you use the `file =` argument, `dput()` can write this representation to file but you won't be tempted to actually read that thing. `dput()` creates an R-specific-but-not-binary representation. Let's try it out.

```
## first restore gap_life_exp with our desired country factor level order
gap_life_exp <- readRDS("gap_life_exp.rds")
dput(gap_life_exp, "gap_life_exp-dput.txt")
```

Now let's look at the first few lines of the file `gap_life_exp-dput.txt`.

```
structure(list(country = structure(c(3L, 107L, 74L, 2L, 98L,
138L, 128L, 102L, 49L, 125L, 26L, 56L, 96L, 47L, 75L, 85L, 18L,
12L, 37L, 24L, 133L, 13L, 16L, 117L, 84L, 82L, 53L, 9L, 28L,
120L, 22L, 104L, 114L, 109L, 115L, 23L, 73L, 97L, 66L, 71L, 15L,
29L, 20L, 122L, 134L, 40L, 35L, 123L, 38L, 126L, 60L, 25L, 7L,
39L, 59L, 141L, 86L, 140L, 51L, 63L, 64L, 52L, 121L, 135L, 132L,
```

Huh? Don't worry about it. Remember we are "writing data for computers". The partner function `dget()` reads this representation back in.

```
gap_life_exp_dget <- dget("gap_life_exp-dput.txt")
country_levels <- country_levels %>%
  mutate(via_dput = head(levels(gap_life_exp_dget$country)))
country_levels
#> # A tibble: 6 x 4
#>   original     via_csv     via_rds      via_dput
#>   <chr>        <chr>       <chr>        <chr>
#> 1 Sierra Leone Afghanistan Sierra Leone Sierra Leone
#> 2 Angola       Albania     Angola       Angola
#> 3 Afghanistan  Algeria     Afghanistan  Afghanistan
#> 4 Liberia      Angola      Liberia      Liberia
#> 5 Rwanda       Argentina   Rwanda       Rwanda
#> 6 Mozambique   Australia   Mozambique   Mozambique
```

Note how the original, post-reordering country factor levels are restored using the `dput()` / `dget()` strategy.

But why on earth would you ever do this?

The main application of this is the creation of highly portable, self-contained minimal examples. For example, if you want to pose a question on a forum or directly to an expert, it might be required or just plain courteous to NOT attach any data files. You will need a monolithic, plain text blob that defines any necessary objects and has the necessary code. `dput()` can be helpful for producing the piece of code that defines the object. If you `dput()` without specifying a file, you can copy the return value from Console and paste into a script. Or you can write to file and copy from there or add R commands below.

9.12 Other types of objects to use `dput()` or `saveRDS()` on

My special dispensation to abandon human-readable, plain text files is even broader than I've let on. Above, I give my blessing to store data.frames via `dput()` and/or `saveRDS()`, when you've done some rational factor level re-ordering. The same advice and mechanics apply a bit more broadly: you're also allowed to use R-specific file formats to save vital non-rectangular objects, such as a fitted nonlinear mixed effects model or a classification and regression tree.

9.13 Clean up

We've written several files in this tutorial. Some of them are not of lasting value or have confusing filenames. I choose to delete them, while demonstrating some of the many functions R offers for interacting with the filesystem. It's up to you whether you want to submit this command or not.

```
file.remove(list.files(pattern = "^gap_life_exp"))
#> [1] TRUE TRUE TRUE
```

9.14 Pitfalls of delimited files

If a delimited file contains fields where a human being has typed, be crazy paranoid because people do really nutty things. Especially people who aren't in the business of programming and have never had to compute on text. Claim: a person's regular expression skill is inversely proportional to the skill required to handle the files they create. Implication: if someone has never heard of regular expressions, prepare for lots of pain working with their files.

When the header fields (often, but not always, the variable names) or actual data contain the delimiter, it can lead to parsing and import failures. Two popular delimiters are the comma `,` and the TAB `\t` and humans tend to use these when typing. If you can design this problem away during data capture, such as by using a drop down menu on an input form, by all means do so. Sometimes this is impossible or undesirable and you must deal with fairly free form text. That's a good time to allow/force text to be protected with quotes, because it will make parsing the delimited file go more smoothly.

Sometimes, instead of rigid tab-delimiting, whitespace is used as the delimiter. That is, in fact, the default for both `read.table()` and `write.table()`. Assuming you will write/read variable names from the first line (a.k.a. the `header` in `write.table()` and `read.table()`), they must be valid R variable names … or they will be coerced into something valid. So, for these two reasons, it is good practice to use "one word" variable names whenever possible. If you need to evoke multiple words, use `snake_case` or `camelCase` to cope. Example: the header entry for the field holding the subject's last name should be `last_name` or `lastName` NOT `last name`. With the readr package, "column names are left as is, not munged into valid R identifiers (i.e. there is no `check.names = TRUE`)". So you can get away with whitespace in variable names and yet I recommend that you do not.

9.15 Resources

Data import chapter of R for Data Science by Hadley Wickham and Garrett Grolemund (2016).

White et al.'s "Nine simple ways to make it easier to (re)use your data" (2013).

- First appeared in PeerJ Preprints
- Published in Ideas in Ecology and Evolution in 2013
- Section 4 "Use Standard Data Formats" is especially good reading.

Wickham's paper on tidy data in the Journal of Statistical Software (2014).

- Available as a PDF here

Data Manipulation in R by Phil Spector (2008).

- Available via SpringerLink
- Author's webpage
- GoogleBooks search
- See Chapter 2 ("Reading and Writing Data")

Part IV

Data analysis 2

Chapter 10

Be the boss of your factors

10.1 Factors: where they fit in

We've spent a lot of time working with big, beautiful data frames, like the Gapminder data. But we also need to manage the individual variables housed within.

Factors are the variable type that useRs love to hate. It is how we store truly categorical information in R. The values a factor can take on are called the **levels**. For example, the levels of the factor `continent` in Gapminder are are "Africa", "Americas", etc. and this is what's usually presented to your eyeballs by R. In general, the levels are friendly human-readable character strings, like "male/female" and "control/treated". But *never ever ever* forget that, under the hood, R is really storing integer codes 1, 2, 3, etc.

This Janus-like nature of factors means they are rich with booby traps for the unsuspecting but they are a necessary evil. I recommend you learn how to be the boss of your factors. The pros far outweigh the cons. Specifically in modelling and figure-making, factors are anticipated and accommodated by the functions and packages you will want to exploit.

The worst kind of factor is the stealth factor. The variable that you think of as character, but that is actually a factor (numeric!!). This is a classic R gotcha. Check your variable types explicitly when things seem weird. It happens to the best of us.

Where do stealth factors come from? Base R has a burning desire to turn character information into factor. The happens most commonly at data import via `read.table()` and friends. But `data.frame()` and other functions are also eager to convert character to factor. To shut this down, use `stringsAsFactors = FALSE` in `read.table()` and `data.frame()` or – even better – **use the tidy-**

verse! For data import, use `readr::read_csv()`, `readr::read_tsv()`, etc. For data frame creation, use `tibble::tibble()`. And so on.

Good articles about how the factor fiasco came to be:

- "stringsAsFactors: An unauthorized biography" by Roger Peng
- "stringsAsFactors = <sigh>" by Thomas Lumley

10.2 The forcats package

forcats is a core package in the tidyverse. It is installed via `install.packages("tidyverse")`, and loaded with `library(tidyverse)`. You can also install via `install.packages("forcats")`and load it yourself separately as needed via `library(forcats)`. Main functions start with `fct_`. There really is no coherent family of base functions that forcats replaces – that's why it's such a welcome addition.

Currently this lesson will be mostly code vs. prose. See the previous lesson for more discussion during the transition.

10.3 Load forcats and gapminder

I choose to load the tidyverse, which will load forcats, among other packages we use incidentally below.

```
library(tidyverse)
#>   Attaching packages                 tidyverse 1.3.0
#>   ggplot2 3.3.2      purrr   0.3.4
#>   tibble  3.0.3      dplyr   1.0.2
#>   tidyr   1.1.2      stringr 1.4.0
#>   readr   1.3.1      forcats 0.5.0
#>   Conflicts                    tidyverse_conflicts()
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Also load gapminder.

```
library(gapminder)
```

10.4 Factor inspection

Get to know your factor before you start touching it! It's polite. Let's use `gapminder$continent` as our example.

```
str(gapminder$continent)
#>  Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
levels(gapminder$continent)
#> [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
nlevels(gapminder$continent)
#> [1] 5
class(gapminder$continent)
#> [1] "factor"
```

To get a frequency table as a tibble, from a tibble, use `dplyr::count()`. To get similar from a free-range factor, use `forcats::fct_count()`.

```
gapminder %>%
  count(continent)
#> # A tibble: 5 x 2
#>   continent     n
#>   <fct>     <int>
#> 1 Africa      624
#> 2 Americas    300
#> 3 Asia        396
#> 4 Europe      360
#> 5 Oceania      24
fct_count(gapminder$continent)
#> # A tibble: 5 x 2
#>   f            n
#>   <fct>    <int>
#> 1 Africa     624
#> 2 Americas   300
#> 3 Asia       396
#> 4 Europe     360
#> 5 Oceania     24
```

10.5 Dropping unused levels

Just because you drop all the rows corresponding to a specific factor level, the levels of the factor itself do not change. Sometimes all these unused levels can come back to haunt you later, e.g., in figure legends.

Watch what happens to the levels of `country` (= nothing) when we filter Gapminder to a handful of countries.

```
nlevels(gapminder$country)
#> [1] 142
```

```
h_countries <- c("Egypt", "Haiti", "Romania", "Thailand", "Venezuela")
h_gap <- gapminder %>%
  filter(country %in% h_countries)
nlevels(h_gap$country)
#> [1] 142
```

Even though `h_gap` only has data for a handful of countries, we are still schlepping around all 142 levels from the original `gapminder` tibble.

How can you get rid of them? The base function `droplevels()` operates on all the factors in a data frame or on a single factor. The function `forcats::fct_drop()` operates on a factor.

```
h_gap_dropped <- h_gap %>%
  droplevels()
nlevels(h_gap_dropped$country)
#> [1] 5

## use forcats::fct_drop() on a free-range factor
h_gap$country %>%
  fct_drop() %>%
  levels()
#> [1] "Egypt"     "Haiti"     "Romania"   "Thailand"  "Venezuela"
```

Exercise: Filter the gapminder data down to rows where population is less than a quarter of a million, i.e. 250,000. Get rid of the unused factor levels for `country` and `continent` in different ways, such as:

- `droplevels()`
- `fct_drop()` inside `mutate()`
- `fct_dopr()` with `mutate_at()` or `mutate_if()`

10.6 Change order of the levels, principled

By default, factor levels are ordered alphabetically. Which might as well be random, when you think about it! It is preferable to order the levels according to some principle:

- Frequency. Make the most common level the first and so on.
- Another variable. Order factor levels according to a summary statistic for another variable. Example: order Gapminder countries by life expectancy.

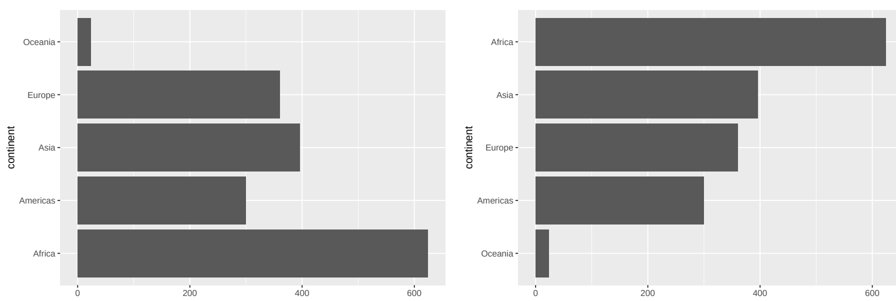
First, let's order continent by frequency, forwards and backwards. This is often a great idea for tables and figures, especially frequency barplots.

```
## default order is alphabetical
gapminder$continent %>%
  levels()
#> [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"

## order by frequency
gapminder$continent %>%
  fct_infreq() %>%
  levels()
#> [1] "Africa"   "Asia"     "Europe"   "Americas" "Oceania"

## backwards!
gapminder$continent %>%
  fct_infreq() %>%
  fct_rev() %>%
  levels()
#> [1] "Oceania"  "Americas" "Europe"   "Asia"     "Africa"
```

These two barcharts of frequency by continent differ only in the order of the continents. Which do you prefer?



Now we order `country` by another variable, forwards and backwards. This other variable is usually quantitative and you will order the factor according to a grouped summary. The factor is the grouping variable and the default summarizing function is `median()` but you can specify something else.

```
## order countries by median life expectancy
fct_reorder(gapminder$country, gapminder$lifeExp) %>%
  levels() %>% head()
#> [1] "Sierra Leone"  "Guinea-Bissau" "Afghanistan"   "Angola"
#> [5] "Somalia"       "Guinea"
```

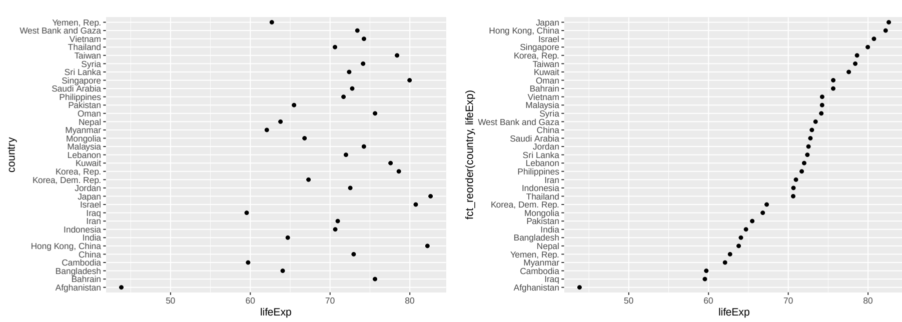
```r
## order accoring to minimum life exp instead of median
fct_reorder(gapminder$country, gapminder$lifeExp, min) %>%
  levels() %>% head()
#> [1] "Rwanda"       "Afghanistan"  "Gambia"       "Angola"
#> [5] "Sierra Leone" "Cambodia"

## backwards!
fct_reorder(gapminder$country, gapminder$lifeExp, .desc = TRUE) %>%
  levels() %>% head()
#> [1] "Iceland"     "Japan"       "Sweden"      "Switzerland"
#> [5] "Netherlands" "Norway"
```

Example of why we reorder factor levels: often makes plots much better! When a factor is mapped to x or y, it should almost always be reordered by the quantitative variable you are mapping to the other one.

Compare the interpretability of these two plots of life expectancy in Asian countries in 2007. The *only difference* is the order of the `country` factor. Which one do you find easier to learn from?

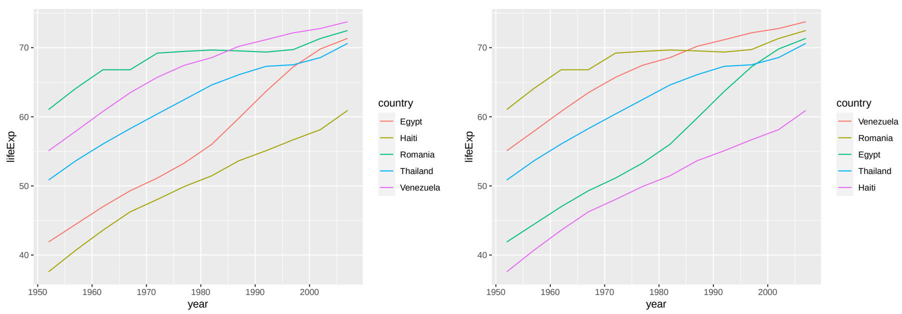
```r
gap_asia_2007 <- gapminder %>% filter(year == 2007, continent == "Asia")
ggplot(gap_asia_2007, aes(x = lifeExp, y = country)) + geom_point()
ggplot(gap_asia_2007, aes(x = lifeExp, y = fct_reorder(country, lifeExp))) +
  geom_point()
```



Use `fct_reorder2()` when you have a line chart of a quantitative x against another quantitative y and your factor provides the color. This way the legend appears in some order as the data! Contrast the legend on the left with the one on the right.

```r
h_countries <- c("Egypt", "Haiti", "Romania", "Thailand", "Venezuela")
h_gap <- gapminder %>%
  filter(country %in% h_countries) %>%
  droplevels()
```

```
ggplot(h_gap, aes(x = year, y = lifeExp, color = country)) +
  geom_line()
ggplot(h_gap, aes(x = year, y = lifeExp,
                  color = fct_reorder2(country, year, lifeExp))) +
  geom_line() +
  labs(color = "country")
```



10.7 Change order of the levels, “because I said so”

Sometimes you just want to hoist one or more levels to the front. Why? Because I said so. This resembles what we do when we move variables to the front with `dplyr::select(special_var, everything())`.

```
h_gap$country %>% levels()
#> [1] "Egypt"     "Haiti"     "Romania"   "Thailand"  "Venezuela"
h_gap$country %>% fct_relevel("Romania", "Haiti") %>% levels()
#> [1] "Romania"   "Haiti"     "Egypt"     "Thailand"  "Venezuela"
```

This might be useful if you are preparing a report for, say, the Romanian government. The reason for always putting Romania first has nothing to do with the *data*, it is important for external reasons and you need a way to express this.

10.8 Recode the levels

Sometimes you have better ideas about what certain levels should be. This is called recoding.

```
i_gap <- gapminder %>%
  filter(country %in% c("United States", "Sweden", "Australia")) %>%
  droplevels()
i_gap$country %>% levels()
#> [1] "Australia"     "Sweden"        "United States"
i_gap$country %>%
  fct_recode("USA" = "United States", "Oz" = "Australia") %>% levels()
#> [1] "Oz"     "Sweden" "USA"
```

Exercise: Isolate the data for `"Australia"`, `"Korea, Dem. Rep."`, and `"Korea, Rep."` in the 2000x. Revalue the country factor levels to `"Oz"`, `"North Korea"`, and `"South Korea"`.

10.9 Grow a factor

Let's create two data frames, each with data from two countries, dropping unused factor levels.

```
df1 <- gapminder %>%
  filter(country %in% c("United States", "Mexico"), year > 2000) %>%
  droplevels()
df2 <- gapminder %>%
  filter(country %in% c("France", "Germany"), year > 2000) %>%
  droplevels()
```

The `country` factors in `df1` and `df2` have different levels.

```
levels(df1$country)
#> [1] "Mexico"        "United States"
levels(df2$country)
#> [1] "France"  "Germany"
```

Can you just combine them?

```
c(df1$country, df2$country)
#> [1] 1 1 2 2 1 1 2 2
```

Umm, no. That is wrong on many levels! Use `fct_c()` to do this.

```
fct_c(df1$country, df2$country)
#> [1] Mexico        Mexico        United States United States
#> [5] France        France        Germany       Germany
#> Levels: Mexico United States France Germany
```

Exercise: Explore how different forms of row binding work behave here, in terms of the `country` variable in the result.

```
bind_rows(df1, df2)
#> # A tibble: 8 x 6
#>   country       continent  year lifeExp       pop gdpPercap
#>   <fct>         <fct>     <int>   <dbl>     <int>     <dbl>
#> 1 Mexico        Americas   2002    74.9 102479927    10742.
#> 2 Mexico        Americas   2007    76.2 108700891    11978.
#> 3 United States Americas   2002    77.3 287675526    39097.
#> 4 United States Americas   2007    78.2 301139947    42952.
#> 5 France        Europe     2002    79.6  59925035    28926.
#> 6 France        Europe     2007    80.7  61083916    30470.
#> 7 Germany       Europe     2002    78.7  82350671    30036.
#> 8 Germany       Europe     2007    79.4  82400996    32170.
rbind(df1, df2)
#> # A tibble: 8 x 6
#>   country       continent  year lifeExp       pop gdpPercap
#>   <fct>         <fct>     <int>   <dbl>     <int>     <dbl>
#> 1 Mexico        Americas   2002    74.9 102479927    10742.
#> 2 Mexico        Americas   2007    76.2 108700891    11978.
#> 3 United States Americas   2002    77.3 287675526    39097.
#> 4 United States Americas   2007    78.2 301139947    42952.
#> 5 France        Europe     2002    79.6  59925035    28926.
#> 6 France        Europe     2007    80.7  61083916    30470.
#> 7 Germany       Europe     2002    78.7  82350671    30036.
#> 8 Germany       Europe     2007    79.4  82400996    32170.
```

Chapter 11

Character vectors

11.1 Character vectors: where they fit in

We've spent a lot of time working with big, beautiful data frames. That are clean and wholesome, like the Gapminder data.

But real life will be much nastier. You will bring data into R from the outside world and discover there are problems. You might think: how hard can it be to deal with character data? And the answer is: it can be very hard!

- Stack Exchange outage
- Regexes to validate/match email addresses
- Fixing an Atom bug

Here we discuss common remedial tasks for cleaning and transforming character data, also known as "strings". A data frame or tibble will consist of one or more *atomic vectors* of a certain class. This lesson deals with things you can do with vectors of class `character`.

11.2 Resources

I start with this because we cannot possibly do this topic justice in a short amount of time. Our goal is to make you aware of broad classes of problems and their respective solutions. Once you have a character problem in real life, these resources will be extremely helpful as you delve deeper.

11.2.1 Manipulating character vectors

- stringr package.
  - A core package in the tidyverse. It is installed via `install.packages("tidyverse")` and also loaded via `library(tidyverse)`. Of course, you can also install or load it individually.
  - Main functions start with `str_`. Auto-complete is your friend.
  - Replacements for base functions re: string manipulation and regular expressions (see below).
  - Main advantages over base functions: greater consistency about inputs and outputs. Outputs are more ready for your next analytical task.
- tidyr package.
  - Especially useful for functions that split one character vector into many and *vice versa*: `separate()`, `unite()`, `extract()`.
- Base functions: `nchar()`, `strsplit()`, `substr()`, `paste()`, `paste0()`.
- The glue package is fantastic for string interpolation. If `stringr::str_interp()` doesn't get your job done, check out the glue package.

11.2.2 Regular expressions resources

A God-awful and powerful language for expressing patterns to match in text or for search-and-replace. Frequently described as "write only", because regular expressions are easier to write than to read/understand. And they are not particularly easy to write.

- We again prefer the stringr package over base functions. Why?
  - Wraps stringi, which is a great place to look if stringr isn't powerful enough.
  - Standardized on ICU regular expressions, so you can stop toggling `perl = TRUE/FALSE` at random.
  - Results come back in a form that is much friendlier for downstream work.
- The Strings chapter of R for Data Science (Wickham and Grolemund, 2016) is a great resource.
- Older STAT 545 lessons on regular expressions have some excellent content. This lesson draws on them, but makes more rigorous use of stringr and uses example data that is easier to support long-term.
  - 2014 Intro to regular expressions by TA Gloria Li (Appendix **??**).
  - 2015 Regular expressions and character data in R by TA Kieran Samuk (Appendix **??**).

- Regular Expressions in R Cheat Sheet by Ian Kopacka.
- Regex testers:
  - regex101.com
  - regexr.com
- rex R package: make regular expression from human readable expressions.
- Base functions: `grep()` and friends.

11.2.3 Character encoding resources

- Strings subsection of data import chapter in R for Data Science (Wickham and Grolemund, 2016).
- Screeds on the Minimum Everyone Needs to Know about encoding:
  - "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)"
  - "What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text"
- Chapter 12 - I've translated this blog post, "3 Steps to Fix Encoding Problems in Ruby", into R as the first step to developing a lesson.

11.2.4 Character vectors that live in a data frame

- Certain operations are facilitated by tidyr. These are described below.
- For a general discussion of how to work on variables that live in a data frame, see Vectors versus tibbles (Appendix ??).

11.3 Load the tidyverse, which includes stringr

```
library(tidyverse)
#>  Attaching packages                tidyverse 1.3.0
#>  ggplot2 3.3.2      purrr   0.3.4
#>  tibble  3.0.3      dplyr   1.0.2
#>  tidyr   1.1.2      stringr 1.4.0
#>  readr   1.3.1      forcats 0.5.0
#>  Conflicts                  tidyverse_conflicts()
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

11.4 Regex-free string manipulation with stringr and tidyr

Basic string manipulation tasks:

- Study a single character vector
  - How long are the strings?
  - Presence/absence of a literal string
- Operate on a single character vector
  - Keep/discard elements that contain a literal string
  - Split into two or more character vectors using a fixed delimiter
  - Snip out pieces of the strings based on character position
  - Collapse into a single string
- Operate on two or more character vectors
  - Glue them together element-wise to get a new character vector.

`fruit`, `words`, and `sentences` are character vectors that ship with stringr for practicing.

11.4.1 Detect or filter on a target string

Determine presence/absence of a literal string with `str_detect()`. Spoiler: later we see `str_detect()` also detects regular expressions.

Which fruits actually use the word "fruit"?

```
str_detect(fruit, pattern = "fruit")
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [11] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [21] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
#> [31] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
#> [41] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [51] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
#> [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [71] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
```

What's the easiest way to get the actual fruits that match? Use `str_subset()` to keep only the matching elements. Note we are storing this new vector `my_fruit` to use in later examples!

```
(my_fruit <- str_subset(fruit, pattern = "fruit"))
#> [1] "breadfruit"   "dragonfruit" "grapefruit"   "jackfruit"
#> [5] "kiwi fruit"   "passionfruit" "star fruit"   "ugli fruit"
```

11.4.2 String splitting by delimiter

Use `stringr::str_split()` to split strings on a delimiter. Some of our fruits are compound words, like "grapefruit", but some have two words, like "ugli fruit". Here we split on a single space `" "`, but show use of a regular expression later.

```
str_split(my_fruit, pattern = " ")
#> [[1]]
#> [1] "breadfruit"
#>
#> [[2]]
#> [1] "dragonfruit"
#>
#> [[3]]
#> [1] "grapefruit"
#>
#> [[4]]
#> [1] "jackfruit"
#>
#> [[5]]
#> [1] "kiwi"  "fruit"
#>
#> [[6]]
#> [1] "passionfruit"
#>
#> [[7]]
#> [1] "star"  "fruit"
#>
#> [[8]]
#> [1] "ugli"  "fruit"
```

It's bummer that we get a *list* back. But it must be so! In full generality, split strings must return list, because who knows how many pieces there will be?

If you are willing to commit to the number of pieces, you can use `str_split_fixed()` and get a character matrix. You're welcome!

```
str_split_fixed(my_fruit, pattern = " ", n = 2)
#>      [,1]           [,2]
```

```
#> [1,] "breadfruit"   ""
#> [2,] "dragonfruit"  ""
#> [3,] "grapefruit"   ""
#> [4,] "jackfruit"    ""
#> [5,] "kiwi"         "fruit"
#> [6,] "passionfruit" ""
#> [7,] "star"         "fruit"
#> [8,] "ugli"         "fruit"
```

If the to-be-split variable lives in a data frame, `tidyr::separate()` will split it into 2 or more variables.

```
my_fruit_df <- tibble(my_fruit)
my_fruit_df %>%
  separate(my_fruit, into = c("pre", "post"), sep = " ")
#> Warning: Expected 2 pieces. Missing pieces filled with `NA` in 5 rows
#> [1, 2, 3, 4, 6].
#> # A tibble: 8 x 2
#>   pre          post
#>   <chr>        <chr>
#> 1 breadfruit   <NA>
#> 2 dragonfruit  <NA>
#> 3 grapefruit   <NA>
#> 4 jackfruit    <NA>
#> 5 kiwi         fruit
#> 6 passionfruit <NA>
#> 7 star         fruit
#> 8 ugli         fruit
```

11.4.3 Substring extraction (and replacement) by position

Count characters in your strings with `str_length()`. Note this is different from the length of the character vector itself.

```
length(my_fruit)
#> [1] 8
str_length(my_fruit)
#> [1] 10 11 10  9 10 12 10 10
```

You can snip out substrings based on character position with `str_sub()`.

```
head(fruit) %>%
  str_sub(1, 3)
#> [1] "app" "apr" "avo" "ban" "bel" "bil"
```

The `start` and `end` arguments are vectorised. Example: a sliding 3-character window.

```
tibble(fruit) %>%
  head() %>%
  mutate(snip = str_sub(fruit, 1:6, 3:8))
#> # A tibble: 6 x 2
#>   fruit        snip
#>   <chr>        <chr>
#> 1 apple        "app"
#> 2 apricot      "pri"
#> 3 avocado      "oca"
#> 4 banana       "ana"
#> 5 bell pepper  " pe"
#> 6 bilberry     "rry"
```

Finally, `str_sub()` also works for assignment, i.e. on the left hand side of `<-`.

```
(x <- head(fruit, 3))
#> [1] "apple"   "apricot" "avocado"
str_sub(x, 1, 3) <- "AAA"
x
#> [1] "AAAle"   "AAAicot" "AAAcado"
```

11.4.4 Collapse a vector

You can collapse a character vector of length `n > 1` to a single string with `str_c()`, which also has other uses (see the following section).

```
head(fruit) %>%
  str_c(collapse = ", ")
#> [1] "apple, apricot, avocado, banana, bell pepper, bilberry"
```

11.4.5 Create a character vector by catenating multiple vectors

If you have two or more character vectors of the same length, you can glue them together element-wise, to get a new vector of that length. Here are some … awful smoothie flavors?

```
str_c(fruit[1:4], fruit[5:8], sep = " & ")
#> [1] "apple & bell pepper"   "apricot & bilberry"
#> [3] "avocado & blackberry"  "banana & blackcurrant"
```

Element-wise catenation can be combined with collapsing.

```
str_c(fruit[1:4], fruit[5:8], sep = " & ", collapse = ", ")
#> [1] "apple & bell pepper, apricot & bilberry, avocado & blackberry, banana & blackc
```

If the to-be-combined vectors are variables in a data frame, you can use `tidyr::unite()` to make a single new variable from them.

```
fruit_df <- tibble(
  fruit1 = fruit[1:4],
  fruit2 = fruit[5:8]
)
fruit_df %>%
  unite("flavor_combo", fruit1, fruit2, sep = " & ")
#> # A tibble: 4 x 1
#>   flavor_combo
#>   <chr>
#> 1 apple & bell pepper
#> 2 apricot & bilberry
#> 3 avocado & blackberry
#> 4 banana & blackcurrant
```

11.4.6 Substring replacement

You can replace a pattern with `str_replace()`. Here we use an explicit string-to-replace, but later we revisit with a regular expression.

```
str_replace(my_fruit, pattern = "fruit", replacement = "THINGY")
#> [1] "breadTHINGY"   "dragonTHINGY"  "grapeTHINGY"   "jackTHINGY"
#> [5] "kiwi THINGY"   "passionTHINGY" "star THINGY"   "ugli THINGY"
```

A special case that comes up a lot is replacing `NA`, for which there is `str_replace_na()`.

```
melons <- str_subset(fruit, pattern = "melon")
melons[2] <- NA
melons
#> [1] "canary melon" NA             "watermelon"
str_replace_na(melons, "UNKNOWN MELON")
#> [1] "canary melon"  "UNKNOWN MELON" "watermelon"
```

If the `NA`-afflicted variable lives in a data frame, you can use `tidyr::replace_na()`.

```
tibble(melons) %>%
  replace_na(replace = list(melons = "UNKNOWN MELON"))
#> # A tibble: 3 x 1
#>   melons
#>   <chr>
#> 1 canary melon
#> 2 UNKNOWN MELON
#> 3 watermelon
```

And that concludes our treatment of regex-free manipulations of character data!

11.5 Regular expressions with stringr

Figure 11.1: From [ThePracticalDev](https://twitter.com/ThePracticalDev/status/774309983467016193)

11.5.1 Load gapminder

The country names in the `gapminder` data frame are convenient for examples. Load it now and store the 142 unique country names to the object `countries`.

```
library(gapminder)
countries <- levels(gapminder$country)
```

11.5.2 Characters with special meaning

Frequently your string tasks cannot be expressed in terms of a fixed string, but can be described in terms of a **pattern**. Regular expressions, a.k.a "regexes", are the standard way to specify these patterns. In regexes, specific characters and constructs take on special meaning in order to match multiple strings.

The first metacharacter is the period `.`, which stands for any single character, except a newline (which by the way, is represented by `\n`). The regex `a.b` will match all countries that have an `a`, followed by any single character, followed by `b`. Yes, regexes are case sensitive, i.e. "Italy" does not match.

```
str_subset(countries, pattern = "i.a")
#>  [1] "Argentina"                "Bosnia and Herzegovina"
#>  [3] "Burkina Faso"             "Central African Republic"
#>  [5] "China"                    "Costa Rica"
#>  [7] "Dominican Republic"       "Hong Kong, China"
#>  [9] "Jamaica"                  "Mauritania"
#> [11] "Nicaragua"                "South Africa"
#> [13] "Swaziland"                "Taiwan"
#> [15] "Thailand"                 "Trinidad and Tobago"
```

Notice that `i.a` matches "ina", "ica", "ita", and more.

Anchors can be included to express where the expression must occur within the string. The `^` indicates the beginning of string and `$` indicates the end.

Note how the regex `i.a$` matches many fewer countries than `i.a` alone. Likewise, more elements of `my_fruit` match `d` than `^d`, which requires "d" at string start.

```
str_subset(countries, pattern = "i.a$")
#> [1] "Argentina"              "Bosnia and Herzegovina"
#> [3] "China"                  "Costa Rica"
#> [5] "Hong Kong, China"       "Jamaica"
#> [7] "South Africa"
str_subset(my_fruit, pattern = "d")
#> [1] "breadfruit"  "dragonfruit"
str_subset(my_fruit, pattern = "^d")
#> [1] "dragonfruit"
```

The metacharacter `\b` indicates a **word boundary** and `\B` indicates NOT a word boundary. This is our first encounter with something called "escaping" and right now I just want you at accept that we need to prepend a second backslash to use these sequences in regexes in R. We'll come back to this tedious point later.

```
str_subset(fruit, pattern = "melon")
#> [1] "canary melon" "rock melon"   "watermelon"
str_subset(fruit, pattern = "\\bmelon")
#> [1] "canary melon" "rock melon"
str_subset(fruit, pattern = "\\Bmelon")
#> [1] "watermelon"
```

11.5.3 Character classes

Characters can be specified via classes. You can make them explicitly "by hand" or use some pre-existing ones. The 2014 STAT 545 regex lesson (Appendix **??**) has a good list of character classes. Character classes are usually given inside square brackets, `[]` but a few come up so often that we have a metacharacter for them, such as `\d` for a single digit.

Here we match `ia` at the end of the country name, preceded by one of the characters in the class. Or, in the negated class, preceded by anything but one of those characters.

```
## make a class "by hand"
str_subset(countries, pattern = "[nls]ia$")
#>  [1] "Albania"    "Australia"  "Indonesia"  "Malaysia"   "Mauritania"
#>  [6] "Mongolia"   "Romania"    "Slovenia"   "Somalia"    "Tanzania"
#> [11] "Tunisia"
## use ^ to negate the class
str_subset(countries, pattern = "[^nls]ia$")
#>  [1] "Algeria"      "Austria"      "Bolivia"      "Bulgaria"
#>  [5] "Cambodia"     "Colombia"     "Croatia"      "Ethiopia"
#>  [9] "Gambia"       "India"        "Liberia"      "Namibia"
#> [13] "Nigeria"      "Saudi Arabia" "Serbia"       "Syria"
#> [17] "Zambia"
```

Here we revisit splitting `my_fruit` with two more general ways to match whitespace: the `\s` metacharacter and the POSIX class `[:space:]`. Notice that we must prepend an extra backslash `\` to escape `\s` and the POSIX class has to be surrounded by two sets of square brackets.

```
## remember this?
# str_split_fixed(fruit, pattern = " ", n = 2)
## alternatives
str_split_fixed(my_fruit, pattern = "\\s", n = 2)
```

```
#>      [,1]          [,2]
#> [1,] "breadfruit"   ""
#> [2,] "dragonfruit"  ""
#> [3,] "grapefruit"   ""
#> [4,] "jackfruit"    ""
#> [5,] "kiwi"         "fruit"
#> [6,] "passionfruit" ""
#> [7,] "star"         "fruit"
#> [8,] "ugli"         "fruit"
str_split_fixed(my_fruit, pattern = "[[:space:]]", n = 2)
#>      [,1]          [,2]
#> [1,] "breadfruit"   ""
#> [2,] "dragonfruit"  ""
#> [3,] "grapefruit"   ""
#> [4,] "jackfruit"    ""
#> [5,] "kiwi"         "fruit"
#> [6,] "passionfruit" ""
#> [7,] "star"         "fruit"
#> [8,] "ugli"         "fruit"
```

Let's see the country names that contain punctuation.

```
str_subset(countries, "[[:punct:]]")
#> [1] "Congo, Dem. Rep." "Congo, Rep."      "Cote d'Ivoire"
#> [4] "Guinea-Bissau"    "Hong Kong, China" "Korea, Dem. Rep."
#> [7] "Korea, Rep."      "Yemen, Rep."
```

11.5.4 Quantifiers

You can decorate characters (and other constructs, like metacharacters and classes) with information about how many characters they are allowed to match.

quantifier	meaning	quantifier	meaning
*	0 or more	{n}	exactly n
+	1 or more	{n,}	at least n
?	0 or 1	{,m}	at most m
		{n,m}	between n and m, inclusive

Explore these by inspecting matches for `l` followed by `e`, allowing for various numbers of characters in between.

`l.*e` will match strings with 0 or more characters in between, i.e. any string with an `l` eventually followed by an `e`. This is the most inclusive regex for this example, so we store the result as `matches` to use as a baseline for comparison.

```
(matches <- str_subset(fruit, pattern = "l.*e"))
#>  [1] "apple"             "bell pepper"       "bilberry"
#>  [4] "blackberry"        "blood orange"      "blueberry"
#>  [7] "cantaloupe"        "chili pepper"      "clementine"
#> [10] "cloudberry"        "elderberry"        "huckleberry"
#> [13] "lemon"             "lime"              "lychee"
#> [16] "mulberry"          "olive"             "pineapple"
#> [19] "purple mangosteen" "salal berry"
```

Change the quantifier from `*` to `+` to require at least one intervening character. The strings that no longer match: all have a literal `le` with no preceding `l` and no following `e`.

```
list(match = intersect(matches, str_subset(fruit, pattern = "l.+e")),
     no_match = setdiff(matches, str_subset(fruit, pattern = "l.+e")))
#> $match
#>  [1] "bell pepper"       "bilberry"          "blackberry"
#>  [4] "blood orange"      "blueberry"         "cantaloupe"
#>  [7] "chili pepper"      "clementine"        "cloudberry"
#> [10] "elderberry"        "huckleberry"       "lime"
#> [13] "lychee"            "mulberry"          "olive"
#> [16] "purple mangosteen" "salal berry"
#>
#> $no_match
#> [1] "apple"     "lemon"     "pineapple"
```

Change the quantifier from `*` to `?` to require at most one intervening character. In the strings that no longer match, the shortest gap between `l` and following `e` is at least two characters.

```
list(match = intersect(matches, str_subset(fruit, pattern = "l.?e")),
     no_match = setdiff(matches, str_subset(fruit, pattern = "l.?e")))
#> $match
#>  [1] "apple"             "bilberry"          "blueberry"
#>  [4] "clementine"        "elderberry"        "huckleberry"
#>  [7] "lemon"             "mulberry"          "pineapple"
#> [10] "purple mangosteen"
#>
#> $no_match
#>  [1] "bell pepper"  "blackberry"   "blood orange" "cantaloupe"
#>  [5] "chili pepper" "cloudberry"   "lime"         "lychee"
#>  [9] "olive"        "salal berry"
```

Finally, we remove the quantifier and allow for no intervening characters. The strings that no longer match lack a literal `le`.

```
list(match = intersect(matches, str_subset(fruit, pattern = "le")),
     no_match = setdiff(matches, str_subset(fruit, pattern = "le")))
#> $match
#> [1] "apple"              "clementine"         "huckleberry"
#> [4] "lemon"              "pineapple"          "purple mangosteen"
#>
#> $no_match
#>  [1] "bell pepper"  "bilberry"     "blackberry"   "blood orange"
#>  [5] "blueberry"    "cantaloupe"   "chili pepper" "cloudberry"
#>  [9] "elderberry"   "lime"         "lychee"       "mulberry"
#> [13] "olive"        "salal berry"
```

11.5.5 Escaping

You've probably caught on by now that there are certain characters with special meaning in regexes, including `$ * + . ? [ ] ^ { } | ( ) \`.

What if you really need the plus sign to be a literal plus sign and not a regex quantifier? You will need to *escape* it by prepending a backslash. But wait … there's more! Before a regex is interpreted as a regular expression, it is also interpreted by R as a string. And backslash is used to escape there as well. So, in the end, you need to preprend two backslashes in order to match a literal plus sign in a regex.

This will be more clear with examples!

11.5.5.1 Escapes in plain old strings

Here is routine, non-regex use of backslash `\` escapes in plain vanilla R strings. We intentionally use `cat()` instead of `print()` here.

- To escape quotes inside quotes:

  ```
  cat("Do you use \"airquotes\" much?")
  #> Do you use "airquotes" much?
  ```

  Sidebar: eliminating the need for these escapes is exactly why people use double quotes inside single quotes and *vice versa*.

- To insert newline (`\n`) or tab (`\t`):

```
cat("before the newline\nafter the newline")
#> before the newline
#> after the newline
cat("before the tab\tafter the tab")
#> before the tab        after the tab
```

11.5.5.2 Escapes in regular expressions

Examples of using escapes in regexes to match characters that would otherwise have a special interpretation.

We know several `gapminder` country names contain a period. How do we isolate them? Although it's tempting, the command `str_subset(countries, pattern = ".")` won't work!

```
## cheating using a POSIX class ;)
str_subset(countries, pattern = "[[:punct:]]")
#> [1] "Congo, Dem. Rep." "Congo, Rep."      "Cote d'Ivoire"
#> [4] "Guinea-Bissau"    "Hong Kong, China" "Korea, Dem. Rep."
#> [7] "Korea, Rep."      "Yemen, Rep."
## using two backslashes to escape the period
str_subset(countries, pattern = "\\.")
#> [1] "Congo, Dem. Rep." "Congo, Rep."      "Korea, Dem. Rep."
#> [4] "Korea, Rep."      "Yemen, Rep."
```

A last example that matches an actual square bracket.

```
(x <- c("whatever", "X is distributed U[0,1]"))
#> [1] "whatever"                "X is distributed U[0,1]"
str_subset(x, pattern = "\\[")
#> [1] "X is distributed U[0,1]"
```

11.5.6 Groups and backreferences

Your first use of regex is likely to be simple matching: detecting or isolating strings that match a pattern.

But soon you will want to use regexes to transform the strings in character vectors. That means you need a way to address specific parts of the matching strings and to operate on them.

You can use parentheses inside regexes to define *groups* and you can refer to those groups later with *backreferences*.

For now, this lesson will refer you to other place to read up on this:

- STAT 545 2014 Intro to regular expressions by TA Gloria Li (Appendix **??**).
- The Strings chapter of R for Data Science (Wickham and Grolemund, 2016).

Chapter 12

Character encoding

12.1 Resources

- Strings subsection of data import chapter in R for Data Science (Wickham and Grolemund, 2016).
- Screeds on the Minimum Everyone Needs to Know about encoding:
  - "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)"
  - "What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text"
- Debugging charts:
  - Windows-1252 Characters to UTF-8 Bytes to Latin-1 Characters
- Character inspection:
  - https://apps.timwhitlock.info/unicode/inspect

12.2 Translating two blog posts from Ruby to R

For now, this page walks through these two mini-tutorials (written for Ruby), but translated to R:

- "3 Steps to Fix Encoding Problems in Ruby"
- "How to Get From Theyâ€™re to They're"

Don't expect much creativity from me here. My goal is faithful translation.

12.3 What is an encoding?

Look at the string `"hello!"` in bytes. *Ruby*

```
irb(main):001:0> "hello!".bytes
=> [104, 101, 108, 108, 111, 33]
```

The base function `charToRaw()` "converts a length-one character string to raw bytes. It does so without taking into account any declared encoding". It displays bytes in hexadecimal. Use `as.integer()` to convert to decimal, which is more intuitive and allows us to compare against the Ruby results.

```
charToRaw("hello!")
#> [1] 68 65 6c 6c 6f 21
as.integer(charToRaw("hello!"))
#> [1] 104 101 108 108 111  33
```

Use a character less common in English: *Ruby*

```
irb(main):002:0> "hellõ!".bytes
=> [104, 101, 108, 108, 225, 185, 143, 33]
```

```
charToRaw("hellõ!")
#> [1] 68 65 6c 6c e1 b9 8f 21
as.integer(charToRaw("hellõ!"))
#> [1] 104 101 108 108 225 185 143  33
```

Now we see that it takes more than one byte to represent `"õ"`. Three in fact: [225, 185, 143]. The encoding of a string defines this relationship: encoding is a map between one or more bytes and a displayable character.

Take a look at what a single set of bytes looks like when you try different encodings.

Here's, a string encoded as ISO-8859-1 (also known as "Latin1") with a special character. *Ruby*

```
irb(main):003:0> str = "hellÔ!".encode("ISO-8859-1"); str.encode("UTF-8")
=> "hellÔ!"

irb(main):004:0> str.bytes
=> [104, 101, 108, 108, 212, 33]
```

```r
string_latin <- iconv("hellÔ!", from = "UTF-8", to = "Latin1")
string_latin
#> [1] "hell\xd4!"
charToRaw(string_latin)
#> [1] 68 65 6c 6c d4 21
as.integer(charToRaw(string_latin))
#> [1] 104 101 108 108 212  33
```

We've confirmed that we have the correct bytes (meaning the same as the Ruby example). What would that string look like interpreted as ISO-8859-5 instead? *Ruby*

```
irb(main):005:0> str.force_encoding("ISO-8859-5"); str.encode("UTF-8")
=> "hell !"
```

```r
iconv(string_latin, from = "ISO-8859-5", to = "UTF-8")
#> [1] "hell !"
```

It's garbled, which is your first tip-off to an encoding problem.

Also not all strings can be represented in all encodings: *Ruby*

```
irb(main):006:0> "hi ".encode("Windows-1252")
Encoding::UndefinedConversionError: U+2211 to WINDOWS-1252 in conversion from UTF-8 to WINDOWS-12
 from (irb):61:in `encode'
 from (irb):61
 from /usr/local/bin/irb:11:in `<main>'
```

```r
(string <- "hi ")
#> [1] "hi "
Encoding(string)
#> [1] "UTF-8"
as.integer(charToRaw(string))
#> [1] 104 105 226 136 145
(string_windows <- iconv(string, from = "UTF-8", to = "Windows-1252"))
#> [1] NA
```

In Ruby, apparently that is an error. In R, we just get `NA`. Alternatively, and somewhat like Ruby, you can specify a substitution for non-convertible bytes.

```r
(string_windows <- iconv(string, from = "UTF-8", to = "Windows-1252", sub = "?"))
#> [1] "hi???"
```

In the Ruby post, we've seen 3 string functions so far. Review and note which R function was used in the translation.

- `encode` translates a string to another encoding. We've used `iconv(x, from = "UTF-8", to = <DIFFERENT_ENCODING>)` here.
- `bytes` shows the bytes that make up a string. We've used `charToRaw()`, which returns hexadecimal in R. For the sake of comparison to the Ruby post, I've converted to decimal with `as.integer()`.
- `force_encoding` shows what the input bytes would look like if interpreted by a different encoding. We've used `iconv(x, from = <DIFFERENT_ENCODING>, to = "UTF-8")`.

12.4 A three-step process for fixing encoding bugs

12.4.1 Discover which encoding your string is actually in.

Shhh. Secret: this is encoded as Windows-1252. `\x99` should be the trademark symbol ™. Ruby can guess at the encoding. *Ruby*

```
irb(main):078:0> "hi\x99!".encoding
=> #<Encoding:UTF-8>
```

Ruby's guess is bad. This is not encoded as UTF-8. R admits it doesn't know and stringi's guess is not good.

```
string <- "hi\x99!"
string
#> [1] "hi\x99!"
Encoding(string)
#> [1] "unknown"
stringi::stri_enc_detect(string)
#> [[1]]
#>   Encoding Language Confidence
#> 1 UTF-16BE                 0.1
#> 2 UTF-16LE                 0.1
#> 3   EUC-JP       ja        0.1
#> 4   EUC-KR       ko        0.1
```

Advice given in post is to sleuth it out based on where the data came from. With larger amounts of text, each language's guessing facilities presumably do better than they do here. In real life, all of this advice can prove to be … overly optimistic?

I find it helpful to scrutinize debugging charts and look for the weird stuff showing up in my text. Here's one that shows what UTF-8 bytes look like when erroneously interpreted under Windows-1252 encoding. This phenomenon is known as *mojibake*, which is a delightful word for a super-annoying phenomenon. If it helps, know that the most common encodings are UTF-8, ISO-8859-1 (or Latin1), and Windows-1252, so that really narrows things down.

12.4.2 Decide which encoding you want the string to be

That's easy. UTF-8. Done.

12.4.3 Re-encode your string

```
irb(main):088:0> "hi\x99!".encode("UTF-8", "Windows-1252")
=> "hi !"
```

```r
string_windows <- "hi\x99!"
string_utf8 <- iconv(string_windows, from = "Windows-1252", to = "UTF-8")
Encoding(string_utf8)
#> [1] "UTF-8"
string_utf8
#> [1] "hi !"
```

12.5 How to Get From Theyâ€™re to They're

Moving on to the second blog post now.

12.5.1 Multi-byte characters

Since we need to represent more than 256 characters, not all can be represented by a single byte. Let's look at the curly single quote. *Ruby*

```
irb(main):001:0> "they're".bytes
=> [116, 104, 101, 121, 226, 128, 153, 114, 101]
```

```r
string_curly <- "they're"
charToRaw(string_curly)
#> [1] 74 68 65 79 e2 80 99 72 65
as.integer(charToRaw(string_curly))
#> [1] 116 104 101 121 226 128 153 114 101
length(as.integer(charToRaw(string_curly)))
#> [1] 9
nchar(string_curly)
#> [1] 7
```

The string has 7 characters, but 9 bytes, because we're using 3 bytes to represent the curly single quote. Let's focus just on that. *Ruby*

```
irb(main):002:0> "’".bytes
=> [226, 128, 153]
```

```
charToRaw("’")
#> [1] e2 80 99
as.integer(charToRaw("’"))
#> [1] 226 128 153
length(as.integer(charToRaw("’")))
#> [1] 3
```

One of the most common encoding fiascos you'll see is this: theyâ€™re. Note that the curly single quote has been turned into a 3 character monstrosity. This is no coincidence. Remember those 3 bytes?

This is what happens when you interpret bytes that represent text in the UTF-8 encoding as if it's encoded as Windows-1252. Learn to recognize it. *Ruby*

```
irb(main):003:0> "they’re".force_encoding("Windows-1252").encode("UTF-8")
=> "theyâ€ re"
```

```
(string_mis_encoded <- iconv(string_curly, to = "UTF-8", from = "windows-1252"))
#> [1] "theyâ€ re"
```

Let's assume this little gem is buried in some large file and you don't immediately notice. So this string is interpreted with the wrong encoding, i.e. stored as the wrong bytes, either in an R object or in a file on disk. Now what?

Let's review the original, correct bytes vs. the current, incorrect bytes and print the associated strings.

```
as.integer(charToRaw(string_curly))
#> [1] 116 104 101 121 226 128 153 114 101
as.integer(charToRaw(string_mis_encoded))
#>  [1] 116 104 101 121 195 162 226 130 172 226 132 162 114 101
string_curly
#> [1] "they’re"
string_mis_encoded
#> [1] "theyâ€ re"
```

12.5.2 Encoding repair

How do you fix this? You have to reverse your steps. You have a UTF-8 encoded string. Convert it back to Windows-1252, to get the original bytes. Then re-encode that as UTF-8. *Ruby*

```
irb(main):006:0> "theyâ€ re".encode("Windows-1252").force_encoding("UTF-8")
=> "they're"
```

```
string_mis_encoded
#> [1] "theyâ€ re"
backwards_one <- iconv(string_mis_encoded, from = "UTF-8", to = "Windows-1252")
backwards_one
#> [1] "they're"
Encoding(backwards_one)
#> [1] "unknown"
as.integer(charToRaw(backwards_one))
#> [1] 116 104 101 121 226 128 153 114 101
as.integer(charToRaw(string_curly))
#> [1] 116 104 101 121 226 128 153 114 101
```

Chapter 13

Dates and times

13.1 Date-time vectors: where they fit in

We've spent a lot of time working with big, beautiful data frames. That are clean and wholesome, like the Gapminder data. With crude temporal information like, "THE YEAR IS 1952".

But real life will be much nastier. This information will come to you with much greater precision, reported to the last second or worse, complicated by time zones and daylight savings time idiosyncrasies. Or in some weird format.

Here we discuss common remedial tasks for dealing with date-times.

13.2 Resources

I start with this because we cannot possibly do this topic justice in a short amount of time. Our goal is to make you aware of specific problems and solutions. Once you have a character problem in real life, these resources will be extremely helpful as you delve deeper.

- Dates and times chapter from R for Data Science by Hadley Wickham and Garrett Grolemund (2016).
  - See also the subsection on dates and times in the Data import chapter.
- The lubridate package.
  - On CRAN.
  - On GitHub.

  - Main vignette: Do more with dates and times in R.

- Grolemund and Wickham's paper on lubridate in the Journal of Statistical Software: "Dates and Times Made Easy with lubridate" (2011).
- Exercises to push you to learn lubridate (*posts include links to answers!*)
  - Part 1
  - Part 2
  - Part 3

13.3 Load the tidyverse and lubridate

```
library(tidyverse)
library(lubridate)
```

13.4 Get your hands on some dates or date-times

Use base `Sys.Date()` or lubridate's `today()` to get today's date, without any time.

```
Sys.Date()
#> [1] "2020-10-27"
today()
#> [1] "2020-10-27"
```

They both give you something of class `Date`.

```
str(Sys.Date())
#>  Date[1:1], format: "2020-10-27"
class(Sys.Date())
#> [1] "Date"
str(today())
#>  Date[1:1], format: "2020-10-27"
class(today())
#> [1] "Date"
```

Use base `Sys.time()` or lubridate's `now()` to get RIGHT NOW, meaning the date and the time.

```
Sys.time()
#> [1] "2020-10-27 17:42:38 CET"
now()
#> [1] "2020-10-27 17:42:38 CET"
```

They both give you something of class `POSIXct` in R jargon.

```
str(Sys.time())
#>  POSIXct[1:1], format: "2020-10-27 17:42:38"
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
str(now())
#>  POSIXct[1:1], format: "2020-10-27 17:42:38"
class(now())
#> [1] "POSIXct" "POSIXt"
```

13.5 Get date or date-time from character

One of the main ways dates and date-times come into your life:

http://r4ds.had.co.nz/dates-and-times.html#creating-datetimes#from-strings

13.6 Build date or date-time from parts

Second most common way dates and date-times come into your life:

http://r4ds.had.co.nz/dates-and-times.html#creating-datetimes#from-individual-components

Once you have dates, you might want to edit them in a non-annoying way:

http://r4ds.had.co.nz/dates-and-times.html#setting-components

13.7 Get parts from date or date-time

http://r4ds.had.co.nz/dates-and-times.html#date-time-components#getting-components

13.8 Arithmetic with date or date-time

http://r4ds.had.co.nz/dates-and-times.html#time-spans

13.9 Get character from date or date-time

Eventually you will need to print this stuff in, say, a report.

I always use `format()` but assumed lubridate had something else/better. Am I missing something here? Probably. For now, read the help: `?format.POSIXct`.

Part V

Data analysis 3

Chapter 14

When one tibble is not enough

We've covered many topics on how to manipulate and reshape a single data frame:

- Chapter 5 - Basic care and feeding of data in R
  - Data frames (and tibbles) are awesome.
- Chapter 6 - Introduction to dplyr
  - Filter, select, the pipe.
- Chapter 7 - dplyr functions for a single dataset
  - Single table verbs.
- Chapter 8 - Tidy data using Lord of the Rings
  - Tidy data, tidyr.
  - *This actually kicks off with a row bind operation, discussed below.*

But what if your data arrives in many pieces? There are many good (and bad) reasons why this might happen. How do you get it into one big beautiful tibble? These tasks break down into 3 main classes:

- Bind
- Join
- Lookup

14.1 Typology of data combination tasks

Bind - This is basically smashing ~~rocks~~ tibbles together. You can smash things together row-wise ("row binding") or column-wise ("column binding"). Why do I characterize this as rock-smashing? They're often fairly crude operations, with lots of responsibility falling on the analyst for making sure that the whole enterprise even makes sense.

When row binding, you need to consider the variables in the two tibbles. Do the same variables exist in each? Are they of the same type? Different approaches for row binding have different combinations of flexibility vs. rigidity around these matters.

When column binding, the onus is entirely on the analyst to make sure that the rows are aligned. I would avoid column binding whenever possible. If you can introduce new variables through any other, safer means, do so! By safer, I mean: use a mechanism where the row alignment is correct **by definition**. A proper join is the gold standard. In addition to joins, functions like `dplyr::mutate()` and `tidyr::separate()` can be very useful for forcing yourself to work inside the constraint of a tibble.

Join - Here you designate a variable (or a combination of variables) as a **key**. A row in one data frame gets matched with a row in another data frame because they have the same key. You can then bring information from variables in a secondary data frame into a primary data frame based on this key-based lookup. That description is incredibly oversimplified, but that's the basic idea.

A variety of row- and column-wise operations fit into this framework, which implies there are many different flavors of join. The concepts and vocabulary around joins come from the database world. The relevant functions in dplyr follow this convention and all mention `join`. The most relevant base R function is `merge()`.

Lookup - Lookups are really just a special case of join. But it's a special case worth making for two reasons:

- If you've ever used `LOOKUP()` and friends in Excel, you already have a mental model for this. Let's exploit that!
- Joins are defined in terms of two tables or data frames. But sometimes this task has a "vector" vibe. You might be creating a vector or variable. Or maybe the secondary data source is a named vector. In any case, there's at least one vector in the mix. I call that a lookup.

Let's explore each type of operation with a few examples.

First, let's load the tidyverse (and expose version information).

```
library(tidyverse)
#> ── Attaching packages ─────────────────── tidyverse 1.3.0 ──
#> ✓ ggplot2 3.3.2     ✓ purrr   0.3.4
#> ✓ tibble  3.0.3     ✓ dplyr   1.0.2
#> ✓ tidyr   1.1.2     ✓ stringr 1.4.0
#> ✓ readr   1.3.1     ✓ forcats 0.5.0
#> ── Conflicts ────────────────────── tidyverse_conflicts() ──
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

14.2 Bind

14.2.1 Row binding

We used word count data from the Lord of the Rings trilogy to explore the concept of tidy data in Chapter 8. That kicked off with a quiet, successful row bind. Let's revisit that.

Here's what a perfect row bind of three (untidy!) data frames looks like.

```
fshIP <- tribble(
                         ~Film,    ~Race, ~Female, ~Male,
  "The Fellowship Of The Ring",    "Elf",    1229,   971,
  "The Fellowship Of The Ring", "Hobbit",      14,  3644,
  "The Fellowship Of The Ring",    "Man",       0,  1995
)
rking <- tribble(
                         ~Film,    ~Race, ~Female, ~Male,
      "The Return Of The King",    "Elf",     183,   510,
      "The Return Of The King", "Hobbit",       2,  2673,
      "The Return Of The King",    "Man",     268,  2459
)
ttow <- tribble(
                         ~Film,    ~Race, ~Female, ~Male,
              "The Two Towers",    "Elf",     331,   513,
              "The Two Towers", "Hobbit",       0,  2463,
              "The Two Towers",    "Man",     401,  3589
)
(lotr_untidy <- bind_rows(fship, ttow, rking))
#> # A tibble: 9 x 4
#>   Film                       Race   Female  Male
#>   <chr>                      <chr>   <dbl> <dbl>
#> 1 The Fellowship Of The Ring Elf      1229   971
#> 2 The Fellowship Of The Ring Hobbit     14  3644
```

```
#> 3 The Fellowship Of The Ring Man        0  1995
#> 4 The Two Towers             Elf      331   513
#> 5 The Two Towers             Hobbit     0  2463
#> 6 The Two Towers             Man      401  3589
#> 7 The Return Of The King     Elf      183   510
#> 8 The Return Of The King     Hobbit     2  2673
#> 9 The Return Of The King     Man      268  2459
```

`dplyr::bind_rows()` works like a charm with these very row-bindable data frames! So does base `rbind()` (try it!).

But what if one of the data frames is somehow missing a variable? Let's mangle one and find out.

```
ttow_no_Female <- ttow %>% mutate(Female = NULL)
bind_rows(fship, ttow_no_Female, rking)
#> # A tibble: 9 x 4
#>   Film                       Race   Female  Male
#>   <chr>                      <chr>   <dbl> <dbl>
#> 1 The Fellowship Of The Ring Elf      1229   971
#> 2 The Fellowship Of The Ring Hobbit     14  3644
#> 3 The Fellowship Of The Ring Man         0  1995
#> 4 The Two Towers             Elf        NA   513
#> 5 The Two Towers             Hobbit     NA  2463
#> 6 The Two Towers             Man        NA  3589
#> 7 The Return Of The King     Elf       183   510
#> 8 The Return Of The King     Hobbit      2  2673
#> 9 The Return Of The King     Man       268  2459
rbind(fship, ttow_no_Female, rking)
#> Error in rbind(deparse.level, ...): numbers of columns of arguments do not match
```

We see that `dplyr::bind_rows()` does the row bind and puts `NA` in for the missing values caused by the lack of `Female` data from The Two Towers. Base `rbind()` refuses to row bind in this situation.

I invite you to experiment with other realistic, challenging scenarios, e.g.:

- Change the order of variables. Does row binding match variables by name or position?
- Row bind data frames where the variable `x` is of one type in one data frame and another type in the other. Try combinations that you think should work and some that should not. What actually happens?
- Row bind data frames in which the factor `x` has different levels in one data frame and different levels in the other. What happens?

In conclusion, row binding usually works when it should (especially with `dplyr::bind_rows()`) and usually doesn't when it shouldn't. The biggest risk is being aggravated.

14.2.2 Column binding

Column binding is much more dangerous because it often "works" when it should not. It's **your job** to the rows are aligned and it's all too easy to screw this up.

The data in `gapminder` was originally excavated from 3 messy Excel spreadsheets: one each for life expectancy, population, and GDP per capital. Let's relive some of the data wrangling joy and show a column bind gone wrong.

I create 3 separate data frames, do some evil row sorting, then column bind. There are no errors. The result `gapminder_garbage` sort of looks OK. Univariate summary statistics and exploratory plots will look OK. But I've created complete nonsense!

```
library(gapminder)

life_exp <- gapminder %>%
  select(country, year, lifeExp)

pop <- gapminder %>%
  arrange(year) %>%
  select(pop)

gdp_percap <- gapminder %>%
  arrange(pop) %>%
  select(gdpPercap)

(gapminder_garbage <- bind_cols(life_exp, pop, gdp_percap))
#> # A tibble: 1,704 x 5
#>    country      year lifeExp      pop gdpPercap
#>    <fct>       <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan  1952    28.8  8425333      880.
#>  2 Afghanistan  1957    30.3  1282697      861.
#>  3 Afghanistan  1962    32.0  9279525     2670.
#>  4 Afghanistan  1967    34.0  4232095     1072.
#>  5 Afghanistan  1972    36.1 17876956     1385.
#>  6 Afghanistan  1977    38.4  8691212     2865.
#>  7 Afghanistan  1982    39.9  6927772     1533.
#>  8 Afghanistan  1987    40.8   120447     1738.
#>  9 Afghanistan  1992    41.7 46886859     3021.
#> 10 Afghanistan  1997    41.8  8730405     1890.
#> # … with 1,694 more rows
```

```r
summary(gapminder$lifeExp)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    23.6    48.2    60.7    59.5    70.8    82.6
summary(gapminder_garbage$lifeExp)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    23.6    48.2    60.7    59.5    70.8    82.6
range(gapminder$gdpPercap)
#> [1]    241 113523
range(gapminder_garbage$gdpPercap)
#> [1]    241 113523
```

One last cautionary tale about column binding. This one requires the use of `cbind()` and it's why the tidyverse is generally unwilling to recycle when combining things of different length.

I create a tibble with most of the `gapminder` columns. I create another with the remainder, but filtered down to just one country. I am able to `cbind()` these objects! Why? Because the 12 rows for Canada divide evenly into the 1704 rows of `gapminder`. Note that `dplyr::bind_cols()` refuses to column bind here.

```r
gapminder_mostly <- gapminder %>% select(-pop, -gdpPercap)
gapminder_leftovers_filtered <- gapminder %>%
  filter(country == "Canada") %>%
  select(pop, gdpPercap)

gapminder_nonsense <- cbind(gapminder_mostly, gapminder_leftovers_filtered)
head(gapminder_nonsense, 14)
#>        country continent year lifeExp      pop gdpPercap
#> 1  Afghanistan      Asia 1952    28.8 14785584     11367
#> 2  Afghanistan      Asia 1957    30.3 17010154     12490
#> 3  Afghanistan      Asia 1962    32.0 18985849     13462
#> 4  Afghanistan      Asia 1967    34.0 20819767     16077
#> 5  Afghanistan      Asia 1972    36.1 22284500     18971
#> 6  Afghanistan      Asia 1977    38.4 23796400     22091
#> 7  Afghanistan      Asia 1982    39.9 25201900     22899
#> 8  Afghanistan      Asia 1987    40.8 26549700     26627
#> 9  Afghanistan      Asia 1992    41.7 28523502     26343
#> 10 Afghanistan      Asia 1997    41.8 30305843     28955
#> 11 Afghanistan      Asia 2002    42.1 31902268     33329
#> 12 Afghanistan      Asia 2007    43.8 33390141     36319
#> 13     Albania    Europe 1952    55.2 14785584     11367
#> 14     Albania    Europe 1957    59.3 17010154     12490
```

This data frame isn't obviously wrong, but it is wrong. See how the Canada's population and GDP per capita repeat for each country?

Bottom line: Row bind when you need to, but inspect the results re: coercion. Column bind only if you must and be extremely paranoid.

14.3 Joins in dplyr

Visit Chapter 15 to see concrete examples of all the joins implemented in dplyr, based on comic characters and publishers.

The most recent release of gapminder includes a new data frame, `country_codes`, with country names and ISO codes. Therefore you can also use it to practice joins.

```
gapminder %>%
  select(country, continent) %>%
  group_by(country) %>%
  slice(1) %>%
  left_join(country_codes)
#> Joining, by = "country"
#> # A tibble: 142 x 4
#> # Groups:   country [142]
#>    country     continent iso_alpha iso_num
#>    <chr>       <fct>     <chr>       <int>
#>  1 Afghanistan Asia      AFG             4
#>  2 Albania     Europe    ALB             8
#>  3 Algeria     Africa    DZA            12
#>  4 Angola      Africa    AGO            24
#>  5 Argentina   Americas  ARG            32
#>  6 Australia   Oceania   AUS            36
#>  7 Austria     Europe    AUT            40
#>  8 Bahrain     Asia      BHR            48
#>  9 Bangladesh  Asia      BGD            50
#> 10 Belgium     Europe    BEL            56
#> # … with 132 more rows
```

14.4 Table Lookup

See Chapter 16 for examples.

Chapter 15

Join two tables

Join (a.k.a. merge) two tables: dplyr join cheatsheet with comic characters and publishers.

15.1 Why the cheatsheet

Examples for those of us who don't speak SQL so good. There are lots of Venn diagrams re: SQL joins on the internet, but I wanted R examples. Those diagrams also utterly fail to show what's really going on vis-a-vis rows AND columns.

Other great places to read about joins:

- The dplyr vignette on Two-table verbs.
- The Relational data chapter in R for Data Science (Wickham and Grolemund, 2016). Excellent diagrams.

15.2 The data

Working with two small data frames: `superheroes` and `publishers`.

```
library(tidyverse) ## dplyr provides the join functions

superheroes <- tibble::tribble(
       ~name, ~alignment,  ~gender,          ~publisher,
   "Magneto",      "bad",   "male",            "Marvel",
     "Storm",     "good", "female",            "Marvel",
```

```
  "Mystique",      "bad", "female",            "Marvel",
    "Batman",     "good",   "male",                "DC",
     "Joker",      "bad",   "male",                "DC",
  "Catwoman",      "bad", "female",                "DC",
   "Hellboy",     "good",   "male", "Dark Horse Comics"
  )

publishers <- tibble::tribble(
  ~publisher, ~yr_founded,
        "DC",       1934L,
    "Marvel",       1939L,
     "Image",       1992L
  )
```

Sorry, cheat sheet does not illustrate "multiple match" situations terribly well.

Sub-plot: watch the row and variable order of the join results for a healthy reminder of why it's dangerous to rely on any of that in an analysis.

15.3 inner_join(superheroes, publishers)

> `inner_join(x, y)`: Return all rows from `x` where there are matching values in `y`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combination of the matches are returned. This is a mutating join.

```
(ijsp <- inner_join(superheroes, publishers))
#> Joining, by = "publisher"
#> # A tibble: 6 x 5
#>   name     alignment gender publisher yr_founded
#>   <chr>    <chr>     <chr>  <chr>          <int>
#> 1 Magneto  bad       male   Marvel          1939
#> 2 Storm    good      female Marvel          1939
#> 3 Mystique bad       female Marvel          1939
#> 4 Batman   good      male   DC              1934
#> 5 Joker    bad       male   DC              1934
#> 6 Catwoman bad       female DC              1934
```

We lose Hellboy in the join because, although he appears in `x = superheroes`, his publisher Dark Horse Comics does not appear in `y = publishers`. The join result has all variables from `x = superheroes` plus `yr_founded`, from `y`.

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

inner_join(x = superheroes, y = publishers)

name

alignment

gender

publisher

yr_founded

Magneto

bad

male

Marvel

1939

Storm

good

female

Marvel

1939

Mystique

bad

female

Marvel

1939

Batman

good

male

DC

1934

Joker

bad

male

DC

1934

Catwoman

bad

female

DC

1934

15.4 semi_join(superheroes, publishers)

> `semi_join(x, y)`: Return all rows from `x` where there are matching values in `y`, keeping just columns from `x`. A semi join differs from an inner join because an inner join will return one row of `x` for each matching row of `y`, where a semi join will never duplicate rows of `x`. This is a filtering join.

```
(sjsp <- semi_join(superheroes, publishers))
#> Joining, by = "publisher"
#> # A tibble: 6 x 4
#>   name     alignment gender publisher
#>   <chr>    <chr>     <chr>  <chr>
#> 1 Magneto  bad       male   Marvel
#> 2 Storm    good      female Marvel
#> 3 Mystique bad       female Marvel
#> 4 Batman   good      male   DC
#> 5 Joker    bad       male   DC
#> 6 Catwoman bad       female DC
```

We get a similar result as with `inner_join()` but the join result contains only the variables originally found in `x = superheroes`.

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

semi_join(x = superheroes, y = publishers)

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

15.5 left_join(superheroes, publishers)

> `left_join(x, y)`: Return all rows from `x`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combination of the matches are returned. This is a mutating join.

```
(ljsp <- left_join(superheroes, publishers))
#> Joining, by = "publisher"
#> # A tibble: 7 x 5
#>   name     alignment gender publisher         yr_founded
#>   <chr>    <chr>     <chr>  <chr>                  <int>
#> 1 Magneto  bad       male   Marvel                  1939
#> 2 Storm    good      female Marvel                  1939
#> 3 Mystique bad       female Marvel                  1939
#> 4 Batman   good      male   DC                      1934
#> 5 Joker    bad       male   DC                      1934
#> 6 Catwoman bad       female DC                      1934
#> 7 Hellboy  good      male   Dark Horse Comics         NA
```

We basically get `x = superheroes` back, but with the addition of variable `yr_founded`, which is unique to `y = publishers`. Hellboy, whose publisher does not appear in `y = publishers`, has an `NA` for `yr_founded`.

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

left_join(x = superheroes, y = publishers)

name

alignment

gender

publisher

yr_founded

Magneto

bad

male

Marvel

1939

Storm

good

female

Marvel

1939

Mystique

bad

female

Marvel

1939

Batman

good

male

DC

1934

Joker

bad

male

DC

1934

Catwoman

bad

female

DC

1934

Hellboy

good

male

Dark Horse Comics

NA

15.6 anti_join(superheroes, publishers)

> `anti_join(x, y)`: Return all rows from `x` where there are not matching values in `y`, keeping just columns from `x`. This is a filtering join.

```
(ajsp <- anti_join(superheroes, publishers))
#> Joining, by = "publisher"
#> # A tibble: 1 x 4
#>   name    alignment gender publisher
#>   <chr>   <chr>     <chr>  <chr>
#> 1 Hellboy good      male   Dark Horse Comics
```

We keep **only** Hellboy now (and do not get `yr_founded`).

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

anti_join(x = superheroes, y = publishers)

name

alignment

gender

publisher

Hellboy

good

male

Dark Horse Comics

15.7 inner_join(publishers, superheroes)

> `inner_join(x, y)`: Return all rows from `x` where there are matching values in `y`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combination of the matches are returned. This is a mutating join.

```
(ijps <- inner_join(publishers, superheroes))
#> Joining, by = "publisher"
#> # A tibble: 6 x 5
#>   publisher yr_founded name     alignment gender
#>   <chr>          <int> <chr>    <chr>     <chr>
#> 1 DC              1934 Batman   good      male
#> 2 DC              1934 Joker    bad       male
#> 3 DC              1934 Catwoman bad       female
#> 4 Marvel          1939 Magneto  bad       male
#> 5 Marvel          1939 Storm    good      female
#> 6 Marvel          1939 Mystique bad       female
```

In a way, this does illustrate multiple matches, if you think about it from the `x = publishers` direction. Every publisher that has a match in `y = superheroes` appears multiple times in the result, once for each match. In fact, we're getting the same result as with `inner_join(superheroes, publishers)`, up to variable order (which you should also never rely on in an analysis).

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

inner_join(x = publishers, y = superheroes)

publisher

yr_founded

name

alignment

gender

DC

1934

Batman

good

male

DC

1934

Joker

bad

male

DC

1934

Catwoman

bad

female

Marvel

1939

Magneto

bad

male

Marvel

1939

Storm

good

female

Marvel

1939

Mystique

bad

female

15.8 semi_join(publishers, superheroes)

> semi_join(x, y): Return all rows from x where there are matching values in y, keeping just columns from x. A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x. This is a filtering join.

```
(sjps <- semi_join(x = publishers, y = superheroes))
#> Joining, by = "publisher"
#> # A tibble: 2 x 2
#>   publisher yr_founded
#>   <chr>          <int>
#> 1 DC              1934
#> 2 Marvel          1939
```

Now the effects of switching the `x` and `y` roles is more clear. The result resembles `x = publishers`, but the publisher Image is lost, because there are no observations where `publisher == "Image"` in `y = superheroes`.

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

semi_join(x = publishers, y = superheroes)

publisher

yr_founded

DC

1934

Marvel

1939

15.9 `left_join(publishers, superheroes)`

> `left_join(x, y)`: Return all rows from `x`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combination of the matches are returned. This is a mutating join.

```
(ljps <- left_join(publishers, superheroes))
#> Joining, by = "publisher"
#> # A tibble: 7 x 5
#>   publisher yr_founded name     alignment gender
#>   <chr>          <int> <chr>    <chr>     <chr>
#> 1 DC              1934 Batman   good      male
#> 2 DC              1934 Joker    bad       male
```

```
#> 3 DC            1934 Catwoman bad       female
#> 4 Marvel        1939 Magneto  bad       male
#> 5 Marvel        1939 Storm    good      female
#> 6 Marvel        1939 Mystique bad       female
#> 7 Image         1992 <NA>     <NA>      <NA>
```

We get a similar result as with `inner_join()` but the publisher Image survives in the join, even though no superheroes from Image appear in `y = superheroes`. As a result, Image has `NA`s for `name`, `alignment`, and `gender`.

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

left_join(x = publishers, y = superheroes)

publisher

yr_founded

name

alignment

gender

DC

1934

Batman

good

male

DC

1934

Joker

bad

male

DC

1934

Catwoman

bad

female

Marvel

1939

Magneto

bad

male

Marvel

1939

Storm

good

female

Marvel

1939

Mystique

bad

female

Image

1992

NA

NA

NA

15.10 `anti_join(publishers, superheroes)`

> `anti_join(x, y)`: Return all rows from `x` where there are not matching values in `y`, keeping just columns from `x`. This is a filtering join.

```
(ajps <- anti_join(publishers, superheroes))
#> Joining, by = "publisher"
#> # A tibble: 1 x 2
#>   publisher yr_founded
#>   <chr>          <int>
#> 1 Image           1992
```

We keep **only** publisher Image now (and the variables found in `x = publishers`).

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

anti_join(x = publishers, y = superheroes)

publisher

yr_founded

Image

1992

15.11 full_join(superheroes, publishers)

> `full_join(x, y)`: Return all rows and all columns from both `x` and `y`. Where there are not matching values, returns `NA` for the one missing. This is a mutating join.

```
(fjsp <- full_join(superheroes, publishers))
#> Joining, by = "publisher"
#> # A tibble: 8 x 5
#>   name     alignment gender publisher         yr_founded
#>   <chr>    <chr>     <chr>  <chr>                  <int>
#> 1 Magneto  bad       male   Marvel                  1939
#> 2 Storm    good      female Marvel                  1939
#> 3 Mystique bad       female Marvel                  1939
#> 4 Batman   good      male   DC                      1934
#> 5 Joker    bad       male   DC                      1934
#> 6 Catwoman bad       female DC                      1934
#> 7 Hellboy  good      male   Dark Horse Comics         NA
#> 8 <NA>     <NA>      <NA>   Image                   1992
```

We get all rows of `x = superheroes` plus a new row from `y = publishers`, containing the publisher Image. We get all variables from `x = superheroes` AND all variables from `y = publishers`. Any row that derives solely from one table or the other carries `NA`s in the variables found only in the other table.

superheroes

name

alignment

gender

publisher

Magneto

bad

male

Marvel

Storm

good

female

Marvel

Mystique

bad

female

Marvel

Batman

good

male

DC

Joker

bad

male

DC

Catwoman

bad

female

DC

Hellboy

good

male

Dark Horse Comics

publishers

publisher

yr_founded

DC

1934

Marvel

1939

Image

1992

full_join(x = superheroes, y = publishers)

name

alignment

gender

publisher

yr_founded

Magneto

bad

male

Marvel

1939

Storm

good

female

Marvel

1939

Mystique

bad

female

Marvel

1939

Batman

good

male

DC

1934

Joker

bad

male

DC

1934

Catwoman

bad

female

DC

1934

Hellboy

good

male

Dark Horse Comics

NA

NA

NA

NA

Image

1992

Chapter 16

Table lookup

I try to use dplyr joins for most tasks that combine data from two tibbles. But sometimes you just need good old "table lookup". Party like it's Microsoft Excel `LOOKUP()` time!

16.1 Load gapminder and the tidyverse

```
library(gapminder)
library(tidyverse)
```

16.2 Create mini Gapminder

Work with a tiny subset of Gapminder, `mini_gap`.

```
mini_gap <- gapminder %>%
  filter(country %in% c("Belgium", "Canada", "United States", "Mexico"),
         year > 2000) %>%
  select(-pop, -gdpPercap) %>%
  droplevels()
mini_gap
#> # A tibble: 8 x 4
#>   country       continent  year lifeExp
#>   <fct>         <fct>     <int>   <dbl>
#> 1 Belgium       Europe     2002    78.3
#> 2 Belgium       Europe     2007    79.4
#> 3 Canada        Americas   2002    79.8
```

```
#> 4 Canada        Americas   2007    80.7
#> 5 Mexico        Americas   2002    74.9
#> 6 Mexico        Americas   2007    76.2
#> 7 United States Americas   2002    77.3
#> 8 United States Americas   2007    78.2
```

16.3 Dorky national food example.

Make a lookup table of national foods. Or at least the stereotype. Yes, I have intentionally kept Mexico in mini-Gapminder and neglected to put Mexico here.

```
food <- tribble(
        ~ country,    ~ food,
        "Belgium",  "waffle",
         "Canada", "poutine",
  "United States", "Twinkie"
)
food
#> # A tibble: 3 x 2
#>   country       food
#>   <chr>         <chr>
#> 1 Belgium       waffle
#> 2 Canada        poutine
#> 3 United States Twinkie
```

16.4 Lookup national food

`match(x, table)` reports where the values in the key `x` appear in the lookup variable `table`. It returns positive integers for use as indices. It assumes `x` and `table` are free-range vectors, i.e. there's no implicit data frame on the radar here.

Gapminder's `country` plays the role of the key `x`. It is replicated, i.e. non-unique, in `mini_gap`, but not in `food`, i.e. no country appears more than once `food$country`. FYI `match()` actually allows for multiple matches by only consulting the first.

```
match(x = mini_gap$country, table = food$country)
#> [1]  1  1  2  2 NA NA  3  3
```

In table lookup, there is always a value variable `y` that you plan to index with the `match(x, table)` result. It often lives together with `table` in a data frame;

they should certainly be the same length and synced up with respect to row order.

But first…

I get `x` and `table` backwards some non-negligible percentage of the time. So I store the match indices and index the data frame where `table` lives with it. Add `x` as a column and eyeball-o-metrically assess that all is well.

```
(indices <- match(x = mini_gap$country, table = food$country))
#> [1]  1  1  2  2 NA NA  3  3
add_column(food[indices, ], x = mini_gap$country)
#> # A tibble: 8 x 3
#>   country       food    x
#>   <chr>         <chr>   <fct>
#> 1 Belgium       waffle  Belgium
#> 2 Belgium       waffle  Belgium
#> 3 Canada        poutine Canada
#> 4 Canada        poutine Canada
#> 5 <NA>          <NA>    Mexico
#> 6 <NA>          <NA>    Mexico
#> 7 United States Twinkie United States
#> 8 United States Twinkie United States
```

Once all looks good, do the actual table lookup and, possibly, add the new info to your main table.

```
mini_gap %>%
  mutate(food = food$food[indices])
#> # A tibble: 8 x 5
#>   country       continent  year lifeExp food
#>   <fct>         <fct>     <int>   <dbl> <chr>
#> 1 Belgium       Europe     2002    78.3 waffle
#> 2 Belgium       Europe     2007    79.4 waffle
#> 3 Canada        Americas   2002    79.8 poutine
#> 4 Canada        Americas   2007    80.7 poutine
#> 5 Mexico        Americas   2002    74.9 <NA>
#> 6 Mexico        Americas   2007    76.2 <NA>
#> 7 United States Americas   2002    77.3 Twinkie
#> 8 United States Americas   2007    78.2 Twinkie
```

Of course, if this was really our exact task, we could have used a join!

```
mini_gap %>%
  left_join(food)
#> Joining, by = "country"
```

```
#> # A tibble: 8 x 5
#>   country       continent  year lifeExp food
#>   <chr>         <fct>     <int>   <dbl> <chr>
#> 1 Belgium       Europe     2002    78.3 waffle
#> 2 Belgium       Europe     2007    79.4 waffle
#> 3 Canada        Americas   2002    79.8 poutine
#> 4 Canada        Americas   2007    80.7 poutine
#> 5 Mexico        Americas   2002    74.9 <NA>
#> 6 Mexico        Americas   2007    76.2 <NA>
#> 7 United States Americas   2002    77.3 Twinkie
#> 8 United States Americas   2007    78.2 Twinkie
```

But sometimes you have a substantive reason (or psychological hangup) that makes you prefer the table look up interface.

16.5 World's laziest table lookup

While I'm here, let's demo another standard R trick that's based on indexing by name.

Imagine the table you want to consult isn't even a tibble but is, instead, a named character vector.

```
(food_vec <- setNames(food$food, food$country))
#>       Belgium        Canada United States
#>      "waffle"     "poutine"     "Twinkie"
```

Another way to get the national foods for mini-Gapminder is to simply index `food_vec` with `mini_gap$country`.

```
mini_gap %>%
  mutate(food = food_vec[country])
#> # A tibble: 8 x 5
#>   country       continent  year lifeExp food
#>   <fct>         <fct>     <int>   <dbl> <chr>
#> 1 Belgium       Europe     2002    78.3 waffle
#> 2 Belgium       Europe     2007    79.4 waffle
#> 3 Canada        Americas   2002    79.8 poutine
#> 4 Canada        Americas   2007    80.7 poutine
#> 5 Mexico        Americas   2002    74.9 Twinkie
#> 6 Mexico        Americas   2007    76.2 Twinkie
#> 7 United States Americas   2002    77.3 <NA>
#> 8 United States Americas   2007    78.2 <NA>
```

HOLD ON. STOP. Twinkies aren't the national food of Mexico!?! What went wrong?

Remember `mini_gap$country` is a factor. So when we use it in an indexing context, it's integer nature is expressed. It is pure luck that we get the right foods for Belgium and Canada. Luckily the Mexico - United States situation tipped us off. Here's what we are really indexing `food_vec` by above:

```
unclass(mini_gap$country)
#> [1] 1 1 2 2 3 3 4 4
#> attr(,"levels")
#> [1] "Belgium"       "Canada"        "Mexico"        "United States"
```

To get our desired result, we need to explicitly coerce `mini_gap$country` to character.

```
mini_gap %>%
  mutate(food = food_vec[as.character(country)])
#> # A tibble: 8 x 5
#>   country       continent  year lifeExp food
#>   <fct>         <fct>     <int>   <dbl> <chr>
#> 1 Belgium       Europe     2002    78.3 waffle
#> 2 Belgium       Europe     2007    79.4 waffle
#> 3 Canada        Americas   2002    79.8 poutine
#> 4 Canada        Americas   2007    80.7 poutine
#> 5 Mexico        Americas   2002    74.9 <NA>
#> 6 Mexico        Americas   2007    76.2 <NA>
#> 7 United States Americas   2002    77.3 Twinkie
#> 8 United States Americas   2007    78.2 Twinkie
```

When your key variable is character (and not a factor), you can skip this step.

Part VI

R as a programming language

Chapter 17

R objects and indexing

R objects (beyond data.frames) and indexing.

> "Rigor and clarity are not synonymous" – Larry Wasserman

> "Never hesitate to sacrifice truth for clarity." – Greg Wilson's dad

17.1 Vectors are everywhere

Your garden variety R object is a vector. A single piece of info that you regard as a scalar is just a vector of length 1 and R will cheerfully let you add stuff to it. Square brackets are used for isolating elements of a vector for inspection, modification, etc. This is often called **indexing**. Go through the following code carefully, as it's really rather surprising. BTW, indexing begins at 1 in R, unlike many other languages that index from 0.

```
x <- 3 * 4
x
#> [1] 12
is.vector(x)
#> [1] TRUE
length(x)
#> [1] 1
x[2] <- 100
x
#> [1]  12 100
x[5] <- 3
x
```

```
#> [1]  12 100  NA  NA   3
x[11]
#> [1] NA
x[0]
#> numeric(0)
```

R is built to work with vectors. Many operations are *vectorized*, i.e. by default they will happen component-wise when given a vector as input. Novices often don't internalize or exploit this and they write lots of unnecessary `for` loops.

```
x <- 1:4
## which would you rather write and read?
## the vectorized version ...
(y <- x^2)
#> [1]  1  4  9 16
## or the for loop version?
z <- vector(mode = mode(x), length = length(x))
for(i in seq_along(x)) {
  z[i] <- x[i]^2
}
identical(y, z)
#> [1] TRUE
```

When reading function documentation, keep your eyes peeled for arguments that can be vectors. You'll be surprised how common they are. For example, the mean and standard deviation of random normal variates can be provided as vectors.

```
set.seed(1999)
rnorm(5, mean = 10^(1:5))
#> [1]     10.7    100.0   1001.2  10001.5 100000.1
round(rnorm(5, sd = 10^(0:4)), 2)
#> [1]     0.52    -5.49  -118.56 -1147.28 11607.42
```

This could be awesome in some settings, but dangerous in others, i.e. if you exploit this by mistake and get no warning. This is one of the reasons it's so important to keep close tabs on your R objects: are they what you expect in terms of their flavor and length or dimensions? Check early and check often.

Notice that R also recycles vectors, if they are not the necessary length. You will get a warning if R suspects recycling is unintended, i.e. when one length is not an integer multiple of another, but recycling is silent if it seems like you know what you're doing. Can be a beautiful thing when you're doing this deliberately, but devastating when you don't.

Question: is there a way to turn recycling off? Not that I know of.

```
(y <- 1:3)
#> [1] 1 2 3
(z <- 3:7)
#> [1] 3 4 5 6 7
y + z
#> Warning in y + z: longer object length is not a multiple of shorter
#> object length
#> [1] 4 6 8 7 9
(y <- 1:10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
(z <- 3:7)
#> [1] 3 4 5 6 7
y + z
#>  [1]  4  6  8 10 12  9 11 13 15 17
```

The combine function `c()` is your go-to function for making vectors.

```
str(c("hello", "world"))
#>  chr [1:2] "hello" "world"
str(c(1:3, 100, 150))
#>  num [1:5] 1 2 3 100 150
```

Plain vanilla R objects are called "atomic vectors" and an absolute requirement is that all the bits of info they hold are of the same flavor, i.e. all numeric or logical or character. If that's not already true upon creation, the elements will be coerced to the same flavor, using a "lowest common denominator" approach (usually character). This is another stellar opportunity for you to create an object of one flavor without meaning to do so and to remain ignorant of that for a long time. Check early, check often.

```
(x <- c("cabbage", pi, TRUE, 4.3))
#> [1] "cabbage"          "3.14159265358979" "TRUE"
#> [4] "4.3"
str(x)
#>  chr [1:4] "cabbage" "3.14159265358979" "TRUE" "4.3"
length(x)
#> [1] 4
mode(x)
#> [1] "character"
class(x)
#> [1] "character"
```

The most important atomic vector types are:

- **logical**: TRUE's AND FALSE's, easily coerced into 1's and 0's
- **numeric**: numbers and, yes, integers and double-precision floating point numbers are different but you can live happily for a long time without worrying about this
- **character**

Let's create some simple vectors for more demos below.

```
n <- 8
set.seed(1)
(w <- round(rnorm(n), 2)) # numeric floating point
#> [1] -0.63  0.18 -0.84  1.60  0.33 -0.82  0.49  0.74
(x <- 1:n) # numeric integer
#> [1] 1 2 3 4 5 6 7 8
## another way to accomplish by hand is x <- c(1, 2, 3, 4, 5, 6, 7, 8)
(y <- LETTERS[1:n]) # character
#> [1] "A" "B" "C" "D" "E" "F" "G" "H"
(z <- runif(n) > 0.3) # logical
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

Use `str()` and any other functions you wish to inspect these objects, such as `length()`, `mode()`, `class()`, `is.numeric()`, `is.logical()`, etc. Like the `is.xxx()` family of functions, there are also `as.xxx()` functions you can experiment with.

```
str(w)
#>  num [1:8] -0.63 0.18 -0.84 1.6 0.33 -0.82 0.49 0.74
length(x)
#> [1] 8
is.logical(y)
#> [1] FALSE
as.numeric(z)
#> [1] 1 1 1 1 1 0 1 0
```

17.2 Indexing a vector

We've said, and even seen, that square brackets are used to index a vector. There is great flexibility in what one can put inside the square brackets and it's worth understanding the many options. They are all useful, just in different contexts.

Most common, useful ways to index a vector:

- **Logical vector**: keep elements associated with TRUE's, ditch the FALSE's

- **Vector of positive integers**: specifying the keepers
- **Vector of negative integers**: specifying the losers
- **Character vector**: naming the keepers

```
w
#> [1] -0.63  0.18 -0.84  1.60  0.33 -0.82  0.49  0.74
names(w) <- letters[seq_along(w)]
w
#>     a     b     c     d     e     f     g     h
#> -0.63  0.18 -0.84  1.60  0.33 -0.82  0.49  0.74
w < 0
#>     a     b     c     d     e     f     g     h
#>  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
which(w < 0)
#> a c f
#> 1 3 6
w[w < 0]
#>     a     c     f
#> -0.63 -0.84 -0.82
seq(from = 1, to = length(w), by = 2)
#> [1] 1 3 5 7
w[seq(from = 1, to = length(w), by = 2)]
#>     a     c     e     g
#> -0.63 -0.84  0.33  0.49
w[-c(2, 5)]
#>     a     c     d     f     g     h
#> -0.63 -0.84  1.60 -0.82  0.49  0.74
w[c('c', 'a', 'f')]
#>     c     a     f
#> -0.84 -0.63 -0.82
```

17.3 Lists hold just about anything

Lists are basically über-vectors in R. It's like a vector, but with no requirement that the elements be of the same flavor. In data analysis, you won't make lists very often, at least not consciously, but you should still know about them. Why?

- data.frames are lists! They are a special case where each element is an atomic vector, all having the same length.
- Many functions will return lists to you and you will want to extract goodies from them, such as the p-value for a hypothesis test or the estimated error variance in a regression model

Here we repeat an assignment from above, using `list()` instead of `c()` to combine things and you'll notice that the different flavors of the constituent parts are retained this time.

```
## earlier: a <- c("cabbage", pi, TRUE, 4.3)
(a <- list("cabbage", pi, TRUE, 4.3))
#> [[1]]
#> [1] "cabbage"
#>
#> [[2]]
#> [1] 3.14
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]
#> [1] 4.3
str(a)
#> List of 4
#>  $ : chr "cabbage"
#>  $ : num 3.14
#>  $ : logi TRUE
#>  $ : num 4.3
length(a)
#> [1] 4
mode(a)
#> [1] "list"
class(a)
#> [1] "list"
```

List components can also have names. You can create or change names after a list already exists or this can be integrated into the initial assignment.

```
names(a)
#> NULL
names(a) <- c("veg", "dessert", "myAim", "number")
a
#> $veg
#> [1] "cabbage"
#>
#> $dessert
#> [1] 3.14
#>
#> $myAim
#> [1] TRUE
```

```
#>
#> $number
#> [1] 4.3
a <- list(veg = "cabbage", dessert = pi, myAim = TRUE, number = 4.3)
names(a)
#> [1] "veg"     "dessert" "myAim"   "number"
```

Indexing a list is similar to indexing a vector but it is necessarily more complex. The fundamental issue is this: if you request a single element from the list, do you want a list of length 1 containing only that element or do you want the element itself? For the former (desired return value is a list), we use single square brackets, `[` and `]`, just like indexing a vector. For the latter (desired return value is a single element), we use a dollar sign `$`, which you've already used to get one variable from a data.frame, or double square brackets, `[[` and `]]`.

The "pepper shaker photos" in R for Data Science (Wickham and Grolemund, 2016) are a splendid visual explanation of the different ways to get stuff out of a list. Highly recommended.

> Warning: the rest of this section might make your eyes glaze over. Skip to the next section if you need to; come back later when some list is ruining your day.

A slightly more complicated list will make our demos more educational. Now we really see that the elements can differ in flavor and length.

```
(a <- list(veg = c("cabbage", "eggplant"),
           tNum = c(pi, exp(1), sqrt(2)),
           myAim = TRUE,
           joeNum = 2:6))
#> $veg
#> [1] "cabbage"  "eggplant"
#>
#> $tNum
#> [1] 3.14 2.72 1.41
#>
#> $myAim
#> [1] TRUE
#>
#> $joeNum
#> [1] 2 3 4 5 6
str(a)
#> List of 4
#>  $ veg   : chr [1:2] "cabbage" "eggplant"
```

```
#>  $ tNum  : num [1:3] 3.14 2.72 1.41
#>  $ myAim : logi TRUE
#>  $ joeNum: int [1:5] 2 3 4 5 6
length(a)
#> [1] 4
class(a)
#> [1] "list"
mode(a)
#> [1] "list"
```

Here's are ways to get a single list element:

```
a[[2]] # index with a positive integer
#> [1] 3.14 2.72 1.41
a$myAim # use dollar sign and element name
#> [1] TRUE
str(a$myAim) # we get myAim itself, a length 1 logical vector
#>  logi TRUE
a[["tNum"]] # index with length 1 character vector
#> [1] 3.14 2.72 1.41
str(a[["tNum"]]) # we get tNum itself, a length 3 numeric vector
#>  num [1:3] 3.14 2.72 1.41
iWantThis <- "joeNum" # indexing with length 1 character object
a[[iWantThis]] # we get joeNum itself, a length 5 integer vector
#> [1] 2 3 4 5 6
a[[c("joeNum", "veg")]] # does not work! can't get > 1 elements! see below
#> Error in a[[c("joeNum", "veg")]]: subscript out of bounds
```

A case when one must use the double bracket approach, as opposed to the dollar sign, is when the indexing object itself is an R object; we show that above.

What if you want more than one element? You must index vector-style with single square brackets. Note that the return value will always be a list, unlike the return value from double square brackets, even if you only request 1 element.

```
names(a)
#> [1] "veg"    "tNum"   "myAim"  "joeNum"
a[c("tNum", "veg")] # indexing by length 2 character vector
#> $tNum
#> [1] 3.14 2.72 1.41
#>
#> $veg
#> [1] "cabbage"  "eggplant"
str(a[c("tNum", "veg")]) # returns list of length 2
#> List of 2
```

```
#>  $ tNum: num [1:3] 3.14 2.72 1.41
#>  $ veg : chr [1:2] "cabbage" "eggplant"
a["veg"] # indexing by length 1 character vector
#> $veg
#> [1] "cabbage"  "eggplant"
str(a["veg"])# returns list of length 1
#> List of 1
#>  $ veg: chr [1:2] "cabbage" "eggplant"
length(a["veg"]) # really, it does!
#> [1] 1
length(a["veg"][[1]]) # contrast with length of the veg vector itself
#> [1] 2
```

17.4 Creating a data.frame explicitly

In data analysis, we often import data into data.frame via `read.table()`. But one can also construct a data.frame directly using `data.frame()`.

```
n <- 8
set.seed(1)
(jDat <- data.frame(w = round(rnorm(n), 2),
                    x = 1:n,
                    y = I(LETTERS[1:n]),
                    z = runif(n) > 0.3,
                    v = rep(LETTERS[9:12], each = 2)))
#>       w x y     z v
#> 1 -0.63 1 A  TRUE I
#> 2  0.18 2 B  TRUE I
#> 3 -0.84 3 C  TRUE J
#> 4  1.60 4 D  TRUE J
#> 5  0.33 5 E  TRUE K
#> 6 -0.82 6 F FALSE K
#> 7  0.49 7 G  TRUE L
#> 8  0.74 8 H FALSE L
str(jDat)
#> 'data.frame':        8 obs. of  5 variables:
#>  $ w: num  -0.63 0.18 -0.84 1.6 0.33 -0.82 0.49 0.74
#>  $ x: int  1 2 3 4 5 6 7 8
#>  $ y: 'AsIs' chr  "A" "B" "C" "D" ...
#>  $ z: logi  TRUE TRUE TRUE TRUE TRUE FALSE ...
#>  $ v: chr  "I" "I" "J" "J" ...
mode(jDat)
#> [1] "list"
```

```
class(jDat)
#> [1] "data.frame"
```

> Sidebar: What is `I()`, used when creating the variable `y` in the above data.frame? Short version: it tells R to do something *quite literally*. Here we are protecting the letters from being coerced to factor. We are ensuring we get a character vector. Note we let character-to-factor conversion happen in creating the `v` variable above. More about (foiling) R's determination to convert character data to factor can be found here.

data.frames really are lists! Double square brackets can be used to get individual variables. Single square brackets can be used to get one or more variables, returned as a data.frame (though `subset(..., select = ...))` is how I would more likely do in a data analysis).

```
is.list(jDat) # data.frames are lists
#> [1] TRUE
jDat[[5]] # this works but I prefer ...
#> [1] "I" "I" "J" "J" "K" "K" "L" "L"
jDat$v # using dollar sign and name, when possible
#> [1] "I" "I" "J" "J" "K" "K" "L" "L"
jDat[c("x", "z")] # get multiple variables
#>   x     z
#> 1 1  TRUE
#> 2 2  TRUE
#> 3 3  TRUE
#> 4 4  TRUE
#> 5 5  TRUE
#> 6 6 FALSE
#> 7 7  TRUE
#> 8 8 FALSE
str(jDat[c("x", "z")]) # returns a data.frame
#> 'data.frame':	8 obs. of  2 variables:
#>  $ x: int  1 2 3 4 5 6 7 8
#>  $ z: logi  TRUE TRUE TRUE TRUE TRUE FALSE ...
identical(subset(jDat, select = c(x, z)), jDat[c("x", "z")])
#> [1] TRUE
```

> Question: How do I make a data.frame from a list? It is an absolute requirement that the constituent vectors have the same length, although they can be of different flavors. Assuming you meet that requirement, use `as.data.frame()` to convert.

```
## note difference in the printing of a list vs. a data.frame
(qDat <- list(w = round(rnorm(n), 2),
              x = 1:(n-1), ## <-- LOOK HERE! I MADE THIS VECTOR SHORTER!
              y = I(LETTERS[1:n])))
#> $w
#> [1] -0.62 -2.21  1.12 -0.04 -0.02  0.94  0.82  0.59
#>
#> $x
#> [1] 1 2 3 4 5 6 7
#>
#> $y
#> [1] "A" "B" "C" "D" "E" "F" "G" "H"
as.data.frame(qDat) ## does not work! elements don't have same length!
#> Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments
qDat$x <- 1:n ## fix the short variable x
(qDat <- as.data.frame(qDat)) ## we're back in business
#>       w x y
#> 1 -0.62 1 A
#> 2 -2.21 2 B
#> 3  1.12 3 C
#> 4 -0.04 4 D
#> 5 -0.02 5 E
#> 6  0.94 6 F
#> 7  0.82 7 G
#> 8  0.59 8 H
```

You will encounter weirder situations in which you want to make a data.frame out of a list and there are many tricks. Ask me and we'll beef up this section.

17.5 Indexing arrays, e.g. matrices

Though data.frames are recommended as the default receptacle for rectangular data, there are times when one will store rectangular data as a matrix instead. A matrix is a generalization of an atomic vector and the requirement that all the elements be of the same flavor still holds. General arrays are available in R, where a matrix is an important special case having dimension 2.

Let's make a simple matrix and give it decent row and column names, which we know is a good practice. You'll see familiar or self-explanatory functions below for getting to know a matrix.

```
## don't worry if the construction of this matrix confuses you; just focus on
## the product
jMat <- outer(as.character(1:4), as.character(1:4),
```

```
              function(x, y) {
                paste0('x', x, y)
                })
jMat
#>      [,1]  [,2]  [,3]  [,4]
#> [1,] "x11" "x12" "x13" "x14"
#> [2,] "x21" "x22" "x23" "x24"
#> [3,] "x31" "x32" "x33" "x34"
#> [4,] "x41" "x42" "x43" "x44"
str(jMat)
#>  chr [1:4, 1:4] "x11" "x21" "x31" "x41" ...
class(jMat)
#> [1] "matrix" "array"
mode(jMat)
#> [1] "character"
dim(jMat)
#> [1] 4 4
nrow(jMat)
#> [1] 4
ncol(jMat)
#> [1] 4
rownames(jMat)
#> NULL
rownames(jMat) <- paste0("row", seq_len(nrow(jMat)))
colnames(jMat) <- paste0("col", seq_len(ncol(jMat)))
dimnames(jMat) # also useful for assignment
#> [[1]]
#> [1] "row1" "row2" "row3" "row4"
#>
#> [[2]]
#> [1] "col1" "col2" "col3" "col4"
jMat
#>      col1  col2  col3  col4
#> row1 "x11" "x12" "x13" "x14"
#> row2 "x21" "x22" "x23" "x24"
#> row3 "x31" "x32" "x33" "x34"
#> row4 "x41" "x42" "x43" "x44"
```

Indexing a matrix is very similar to indexing a vector or a list: use square brackets and index with logical, integer numeric (positive or negative), or character vectors. Combine those approaches if you like! The main new wrinkle is the use of a comma `,` to distinguish rows and columns. The i, j-th element is the element at the intersection of row i and column j and is obtained with `jMat[i, j]`. Request an entire row or an entire column by simply leaving the associated index empty. The `drop =` argument controls whether the return value should be an atomic vector (`drop = TRUE`) or a matrix with a single row or column (`drop = FALSE`). Notice how row and column names persist and can help you stay oriented.

```
jMat[2, 3]
#> [1] "x23"
jMat[2, ] # getting row 2
#>  col1  col2  col3  col4
#> "x21" "x22" "x23" "x24"
is.vector(jMat[2, ]) # we get row 2 as an atomic vector
#> [1] TRUE
jMat[ , 3, drop = FALSE] # getting column 3
#>      col3
#> row1 "x13"
#> row2 "x23"
#> row3 "x33"
#> row4 "x43"
dim(jMat[ , 3, drop = FALSE]) # we get column 3 as a 4 x 1 matrix
#> [1] 4 1
jMat[c("row1", "row4"), c("col2", "col3")]
#>      col2  col3
#> row1 "x12" "x13"
#> row4 "x42" "x43"
jMat[-c(2, 3), c(TRUE, TRUE, FALSE, FALSE)] # wacky but possible
#>      col1  col2
#> row1 "x11" "x12"
#> row4 "x41" "x42"
```

Under the hood, of course, matrices are just vectors with some extra facilities for indexing. R is a column-major order language, in contrast to C and Python which use row-major order. What this means is that in the underlying vector storage of a matrix, the columns are stacked up one after the other. Matrices can be indexed *exactly* like a vector, i.e. with no comma i, j business, like so:

```
jMat[7]
#> [1] "x32"
jMat
#>      col1  col2  col3  col4
#> row1 "x11" "x12" "x13" "x14"
#> row2 "x21" "x22" "x23" "x24"
#> row3 "x31" "x32" "x33" "x34"
#> row4 "x41" "x42" "x43" "x44"
```

How to understand this: start counting in the upper left corner, move down the column, continue from the top of column 2 and you'll land on the element "x32" when you get to 7.

If you have meaningful, systematic row or column names, there are many possibilities for indexing via regular expressions. Maybe we will talk about `grep` later….

```
jMat[1, grepl("[24]", colnames(jMat))]
#>  col2  col4
#> "x12" "x14"
```

Note also that one can put an indexed matrix on the receiving end of an assignment operation and, as long as your replacement values have valid shape or extent, you can change the matrix.

```
jMat["row1", 2:3] <- c("HEY!", "THIS IS NUTS!")
jMat
#>      col1  col2   col3            col4
#> row1 "x11" "HEY!" "THIS IS NUTS!" "x14"
#> row2 "x21" "x22"  "x23"           "x24"
#> row3 "x31" "x32"  "x33"           "x34"
#> row4 "x41" "x42"  "x43"           "x44"
```

Note that R can also work with vectors and matrices in the proper mathematical sense, i.e. perform matrix algebra. That is a separate topic. To get you started, read the help on `%*%` for matrix multiplication….

17.6 Creating arrays, e.g. matrices

There are three main ways to create a matrix. It goes without saying that the inputs must comply with the requirement that all matrix elements are the same flavor. If that's not true, you risk an error or, worse, silent conversion to character.

- Filling a matrix with a vector
- Glueing vectors together as rows or columns
- Conversion of a data.frame

Let's demonstrate. Here we fill a matrix with a vector, explore filling by rows and giving row and columns at creation. Notice that recycling happens here too, so if the input vector is not large enough, R will recycle it.

```
matrix(1:15, nrow = 5)
#>      [,1] [,2] [,3]
#> [1,]    1    6   11
```

```
#> [2,]    2    7   12
#> [3,]    3    8   13
#> [4,]    4    9   14
#> [5,]    5   10   15
matrix("yo!", nrow = 3, ncol = 6)
#>      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
#> [1,] "yo!" "yo!" "yo!" "yo!" "yo!" "yo!"
#> [2,] "yo!" "yo!" "yo!" "yo!" "yo!" "yo!"
#> [3,] "yo!" "yo!" "yo!" "yo!" "yo!" "yo!"
matrix(c("yo!", "foo?"), nrow = 3, ncol = 6)
#>      [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
#> [1,] "yo!"  "foo?" "yo!"  "foo?" "yo!"  "foo?"
#> [2,] "foo?" "yo!"  "foo?" "yo!"  "foo?" "yo!"
#> [3,] "yo!"  "foo?" "yo!"  "foo?" "yo!"  "foo?"
matrix(1:15, nrow = 5, byrow = TRUE)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
#> [3,]    7    8    9
#> [4,]   10   11   12
#> [5,]   13   14   15
matrix(1:15, nrow = 5,
       dimnames = list(paste0("row", 1:5),
                       paste0("col", 1:3)))
#>      col1 col2 col3
#> row1    1    6   11
#> row2    2    7   12
#> row3    3    8   13
#> row4    4    9   14
#> row5    5   10   15
```

Here we create a matrix by glueing vectors together. Watch the vector names propagate as row or column names.

```
vec1 <- 5:1
vec2 <- 2^(1:5)
cbind(vec1, vec2)
#>      vec1 vec2
#> [1,]    5    2
#> [2,]    4    4
#> [3,]    3    8
#> [4,]    2   16
#> [5,]    1   32
rbind(vec1, vec2)
#>      [,1] [,2] [,3] [,4] [,5]
```

```
#> vec1    5    4    3    2    1
#> vec2    2    4    8   16   32
```

Here we create a matrix from a data.frame.

```
vecDat <- data.frame(vec1 = 5:1,
                     vec2 = 2^(1:5))
str(vecDat)
#> 'data.frame':        5 obs. of  2 variables:
#>  $ vec1: int  5 4 3 2 1
#>  $ vec2: num  2 4 8 16 32
vecMat <- as.matrix(vecDat)
str(vecMat)
#>  num [1:5, 1:2] 5 4 3 2 1 2 4 8 16 32
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:2] "vec1" "vec2"
```

Here we create a matrix from a data.frame, but experience the "silently convert everything to character" fail. As an added bonus, I'm also allowing the "convert character to factor" thing to happen when we create the data.frame initially. Let this be a reminder to take control of your objects!

```
multiDat <- data.frame(vec1 = 5:1,
                       vec2 = paste0("hi", 1:5))
str(multiDat)
#> 'data.frame':        5 obs. of  2 variables:
#>  $ vec1: int  5 4 3 2 1
#>  $ vec2: chr  "hi1" "hi2" "hi3" "hi4" ...
(multiMat <- as.matrix(multiDat))
#>      vec1 vec2
#> [1,] "5"  "hi1"
#> [2,] "4"  "hi2"
#> [3,] "3"  "hi3"
#> [4,] "2"  "hi4"
#> [5,] "1"  "hi5"
str(multiMat)
#>  chr [1:5, 1:2] "5" "4" "3" "2" ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:2] "vec1" "vec2"
```

17.7 Putting it all together...implications for data.frames

This behind the scenes tour is still aimed at making you a better data analyst. Hopefully the slog through vectors, matrices, and lists will be redeemed by greater prowess at manipulating data.frames. Why should this be true?

- A data.frame is a *list*
- The list elements are the variables and they are *atomic vectors*
- data.frames are rectangular, like their matrix friends, so your intuition – and even some syntax – can be borrowed from the matrix world

A data.frame is a list that quacks like a matrix.

Reviewing list-style indexing of a data.frame:

```
jDat
#>       w x y     z v
#> 1 -0.63 1 A  TRUE I
#> 2  0.18 2 B  TRUE I
#> 3 -0.84 3 C  TRUE J
#> 4  1.60 4 D  TRUE J
#> 5  0.33 5 E  TRUE K
#> 6 -0.82 6 F FALSE K
#> 7  0.49 7 G  TRUE L
#> 8  0.74 8 H FALSE L
jDat$z
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
iWantThis <- "z"
jDat[[iWantThis]]
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
str(jDat[[iWantThis]]) # we get an atomic vector
#>  logi [1:8] TRUE TRUE TRUE TRUE TRUE FALSE ...
```

Reviewing vector-style indexing of a data.frame:

```
jDat["y"]
#>   y
#> 1 A
#> 2 B
#> 3 C
#> 4 D
#> 5 E
#> 6 F
```

```
#> 7 G
#> 8 H
str(jDat["y"]) # we get a data.frame with one variable, y
#> 'data.frame':        8 obs. of  1 variable:
#>  $ y: 'AsIs' chr  "A" "B" "C" "D" ...
iWantThis <- c("w", "v")
jDat[iWantThis] # index with a vector of variable names
#>       w v
#> 1 -0.63 I
#> 2  0.18 I
#> 3 -0.84 J
#> 4  1.60 J
#> 5  0.33 K
#> 6 -0.82 K
#> 7  0.49 L
#> 8  0.74 L
str(jDat[c("w", "v")])
#> 'data.frame':        8 obs. of  2 variables:
#>  $ w: num  -0.63 0.18 -0.84 1.6 0.33 -0.82 0.49 0.74
#>  $ v: chr  "I" "I" "J" "J" ...
str(subset(jDat, select = c(w, v))) # using subset() function
#> 'data.frame':        8 obs. of  2 variables:
#>  $ w: num  -0.63 0.18 -0.84 1.6 0.33 -0.82 0.49 0.74
#>  $ v: chr  "I" "I" "J" "J" ...
```

Demonstrating matrix-style indexing of a data.frame:

```
jDat[ , "v"]
#> [1] "I" "I" "J" "J" "K" "K" "L" "L"
str(jDat[ , "v"])
#>  chr [1:8] "I" "I" "J" "J" ...
jDat[ , "v", drop = FALSE]
#>   v
#> 1 I
#> 2 I
#> 3 J
#> 4 J
#> 5 K
#> 6 K
#> 7 L
#> 8 L
str(jDat[ , "v", drop = FALSE])
#> 'data.frame':        8 obs. of  1 variable:
#>  $ v: chr  "I" "I" "J" "J" ...
jDat[c(2, 4, 7), c(1, 4)] # awful and arbitrary but syntax works
```

```
#>      w    z
#> 2 0.18 TRUE
#> 4 1.60 TRUE
#> 7 0.49 TRUE
jDat[jDat$z, ]
#>       w x y    z v
#> 1 -0.63 1 A TRUE I
#> 2  0.18 2 B TRUE I
#> 3 -0.84 3 C TRUE J
#> 4  1.60 4 D TRUE J
#> 5  0.33 5 E TRUE K
#> 7  0.49 7 G TRUE L
subset(jDat, subset = z)
#>       w x y    z v
#> 1 -0.63 1 A TRUE I
#> 2  0.18 2 B TRUE I
#> 3 -0.84 3 C TRUE J
#> 4  1.60 4 D TRUE J
#> 5  0.33 5 E TRUE K
#> 7  0.49 7 G TRUE L
```

17.8 Table of atomic R object flavors

This table will be hideous unless Pandoc is used to compile.

"flavor"	type reported by typeof()	mode()	class()
character	character	character	character
logical	logical	logical	logical
numeric	integer or double	numeric	integer or double
factor	integer	numeric	factor

This should be legible no matter what.

```
+-----------+---------------+-----------+-----------+
| "flavor"  | type reported | mode()    | class()   |
|           | by typeof()   |           |           |
+===========+===============+===========+===========+
| character | character     | character | character |
+-----------+---------------+-----------+-----------+
```

```
| logical   | logical       | logical   | logical   |
+-----------+---------------+-----------+-----------+
| numeric   | integer       | numeric   | integer   |
|           | or double     |           | or double |
+-----------+---------------+-----------+-----------+
| factor    | integer       | numeric   | factor    |
+-----------+---------------+-----------+-----------+
```

Thinking about objects according to the flavors above will work fairly well for most purposes most of the time, at least when you're first getting started. Notice that most rows in the table are quite homogeneous, i.e. a logical vector is a logical vector is a logical vector. But the row pertaining to factors is an exception, which highlights the special nature of factors. (For more, go here).

Chapter 18

Write your own R functions, part 1

18.1 What and why?

My goal here is to reveal the **process** a long-time useR employs for writing functions. I also want to illustrate why the process is the way it is. Merely looking at the finished product, e.g. source code for R packages, can be extremely deceiving. Reality is generally much uglier … but more interesting!

Why are we covering this now, smack in the middle of data aggregation? Powerful machines like dplyr, purrr, and the built-in "apply" family of functions, are ready and waiting to apply your purpose-built functions to various bits of your data. If you can express your analytical wishes in a function, these tools will give you great power.

18.2 Load the Gapminder data

As usual, load gapminder.

```
library(gapminder)
str(gapminder)
#> tibble [1,704 × 6] (S3: tbl_df/tbl/data.frame)
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 ..
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 ..
#>  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 ..
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
```

```
#>  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 130794..
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

18.3 Max - min

Say you've got a numeric vector, and you want to compute the difference between its max and min. `lifeExp` or `pop` or `gdpPercap` are great examples of a typical input. You can imagine wanting to get this statistic after we slice up the Gapminder data by year, country, continent, or combinations thereof.

18.4 Get something that works

First, develop some working code for interactive use, using a representative input. I'll use Gapminder's life expectancy variable.

R functions that will be useful: `min()`, `max()`, `range()`. (**Read their documentation:** here and here)

```
## get to know the functions mentioned above
min(gapminder$lifeExp)
#> [1] 23.6
max(gapminder$lifeExp)
#> [1] 82.6
range(gapminder$lifeExp)
#> [1] 23.6 82.6

## some natural solutions
max(gapminder$lifeExp) - min(gapminder$lifeExp)
#> [1] 59
with(gapminder, max(lifeExp) - min(lifeExp))
#> [1] 59
range(gapminder$lifeExp)[2] - range(gapminder$lifeExp)[1]
#> [1] 59
with(gapminder, range(lifeExp)[2] - range(lifeExp)[1])
#> [1] 59
diff(range(gapminder$lifeExp))
#> [1] 59
```

Internalize this "answer" because our informal testing relies on you noticing departures from this.

18.4.1 Skateboard » perfectly formed rear-view mirror

This image widely attributed to the Spotify development team conveys an important point.

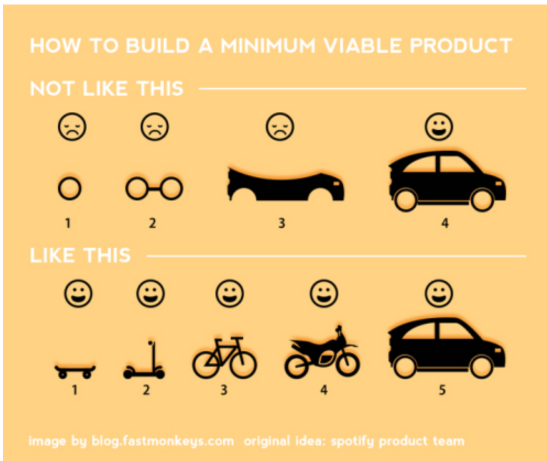


Figure 18.1: From ["Your ultimate guide to Minimum Viable Product (+great examples)"](https://blog.fastmonkeys.com/2014/06/18/minimum-viable-product-your-ultimate-guide-to-mvp-great-examples/)

Build that skateboard before you build the car or some fancy car part. A limited-but-functioning thing is very useful. It also keeps the spirits high.

This is related to the valuable Telescope Rule:

> It is faster to make a four-inch mirror then a six-inch mirror than to make a six-inch mirror.

18.5 Turn the working interactive code into a function

Add NO new functionality! Just write your very first R function.

```
max_minus_min <- function(x) max(x) - min(x)
max_minus_min(gapminder$lifeExp)
#> [1] 59
```

Check that you're getting the same answer as you did with your interactive code. Test it eyeball-o-metrically at this point.

18.6 Test your function

18.6.1 Test on new inputs

Pick some new artificial inputs where you know (at least approximately) what your function should return.

```
max_minus_min(1:10)
#> [1] 9
max_minus_min(runif(1000))
#> [1] 0.997
```

I know that 10 minus 1 is 9. I know that random uniform [0, 1] variates will be between 0 and 1. Therefore max - min should be less than 1. If I take LOTS of them, max - min should be pretty close to 1.

It is intentional that I tested on integer input as well as floating point. Likewise, I like to use valid-but-random data for this sort of check.

18.6.2 Test on real data but *different* real data

Back to the real world now. Two other quantitative variables are lying around: `gdpPercap` and `pop`. Let's have a go.

```
max_minus_min(gapminder$gdpPercap)
#> [1] 113282
max_minus_min(gapminder$pop)
#> [1] 1318623085
```

Either check these results "by hand" or apply the "does that even make sense?" test.

18.6.3 Test on weird stuff

Now we try to break our function. Don't get truly diabolical (yet). Just make the kind of mistakes you can imagine making at 2 a.m. when, 3 years from now, you rediscover this useful function you wrote. Give your function inputs it's not expecting.

```
max_minus_min(gapminder) ## hey sometimes things "just work" on data.frames!
#> Error in FUN(X[[i]], ...): only defined on a data frame with all numeric variables
max_minus_min(gapminder$country) ## factors are kind of like integer vectors, no?
#> Error in Summary.factor(structure(c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, : 'max' not
```

```
max_minus_min("eggplants are purple") ## i have no excuse for this one
#> Error in max(x) - min(x): non-numeric argument to binary operator
```

How happy are you with those error messages? You must imagine that some entire **script** has failed and that you were hoping to just `source()` it without re-reading it. If a colleague or future you encountered these errors, do you run screaming from the room? How hard is it to pinpoint the usage problem?

18.6.4 I will scare you now

Here are some great examples STAT 545 students devised during class where the function **should break but it does not.**

```
max_minus_min(gapminder[c('lifeExp', 'gdpPercap', 'pop')])
#> [1] 1.32e+09
max_minus_min(c(TRUE, TRUE, FALSE, TRUE, TRUE))
#> [1] 1
```

In both cases, R's eagerness to make sense of our requests is unfortunately successful. In the first case, a data.frame containing just the quantitative variables is eventually coerced into numeric vector. We can compute max minus min, even though it makes absolutely no sense at all. In the second case, a logical vector is converted to zeroes and ones, which might merit an error or at least a warning.

18.7 Check the validity of arguments

For functions that will be used again – which is not all of them! – it is good to check the validity of arguments. This implements a rule from the Unix philosophy:

> Rule of Repair: When you must fail, fail noisily and as soon as possible.

18.7.1 stop if not

`stopifnot()` is the entry level solution. I use it here to make sure the input `x` is a numeric vector.

```
mmm <- function(x) {
  stopifnot(is.numeric(x))
  max(x) - min(x)
}
mmm(gapminder)
#> Error in mmm(gapminder): is.numeric(x) is not TRUE
mmm(gapminder$country)
#> Error in mmm(gapminder$country): is.numeric(x) is not TRUE
mmm("eggplants are purple")
#> Error in mmm("eggplants are purple"): is.numeric(x) is not TRUE
mmm(gapminder[c('lifeExp', 'gdpPercap', 'pop')])
#> Error in mmm(gapminder[c("lifeExp", "gdpPercap", "pop")]): is.numeric(x) is not TRU
mmm(c(TRUE, TRUE, FALSE, TRUE, TRUE))
#> Error in mmm(c(TRUE, TRUE, FALSE, TRUE, TRUE)): is.numeric(x) is not TRUE
```

And we see that it catches all of the self-inflicted damage we would like to avoid.

18.7.2 if then stop

`stopifnot()` doesn't provide a very good error message. The next approach is very widely used. Put your validity check inside an `if()` statement and call `stop()` yourself, with a custom error message, in the body.

```
mmm2 <- function(x) {
  if(!is.numeric(x)) {
    stop('I am so sorry, but this function only works for numeric input!\n',
         'You have provided an object of class: ', class(x)[1])
  }
  max(x) - min(x)
}
mmm2(gapminder)
#> Error in mmm2(gapminder): I am so sorry, but this function only works for numeric i
#> You have provided an object of class: tbl_df
```

In addition to a gratuitous apology, the error raised also contains two more pieces of helpful info:

- *Which* function threw the error.
- Hints on how to fix things: expected class of input vs. actual class.

If it is easy to do so, I highly recommend this template: "you gave me THIS, but I need THAT".

The tidyverse style guide has a very useful chapter on how to construct error messages.

18.7.3 Sidebar: non-programming uses for assertions

Another good use of this pattern is to leave checks behind in data analytical scripts. Consider our repetitive use of Gapminder in this course. Every time we load it, we inspect it, hoping to see the usual stuff. If we were loading from file (vs. a stable data package), we might want to formalize our expectations about the number of rows and columns, the names and flavors of the variables, etc. This would alert us if the data suddenly changed, which can be a useful wake-up call in scripts that you re-run *ad nauseam* on auto-pilot or non-interactively.

18.8 Wrap-up and what's next?

Here's the function we've written so far:

```
mmm2
#> function(x) {
#>   if(!is.numeric(x)) {
#>     stop('I am so sorry, but this function only works for numeric input!\n',
#>          'You have provided an object of class: ', class(x)[1])
#>   }
#>   max(x) - min(x)
#> }
```

What we've accomplished:

- We've written our first function.
- We are checking the validity of its input, argument `x`.
- We've done a good amount of informal testing.

Where to next? In part 2 we generalize this function to take differences in other quantiles and learn how to set default values for arguments.

18.9 Resources

- Packages for runtime assertions:
  - assertthat on CRAN and GitHub - *the Hadleyverse option*
  - ensurer on CRAN and GitHub - *general purpose, pipe-friendly*
  - assertr on CRAN and GitHub - *explicitly data pipeline oriented*
  - assertive on CRAN and Bitbucket - *rich set of built-in functions*
- Hadley Wickham's book, Advanced R (2015):
  - Section on defensive programming

Chapter 19

Write your own R functions, part 2

19.1 Where were we? Where are we going?

In part 1 we wrote our first R function to compute the difference between the max and min of a numeric vector. We checked the validity of the function's only argument and, informally, we verified that it worked pretty well.

In this part, we generalize this function, learn more technical details about R functions, and set default values for some arguments.

19.2 Load the Gapminder data

As usual, load gapminder.

```
library(gapminder)
```

19.3 Restore our max minus min function

Let's keep our previous function around as a baseline.

```
mmm <- function(x) {
  stopifnot(is.numeric(x))
  max(x) - min(x)
}
```

19.4 Generalize our function to other quantiles

The max and the min are special cases of a **quantile**. Here are other special cases you may have heard of:

- Median = 0.5 quantile
- 1st quartile = 0.25 quantile
- 3rd quartile = 0.75 quantile

If you're familiar with box plots, the rectangle typically runs from the 1st quartile to the 3rd quartile, with a line at the median.

If q is the p-th quantile of a set of n observations, what does that mean? Approximately pn of the observations are less than q and $(1-p)n$ are greater than q. Yeah, you need to worry about rounding to an integer and less/greater than or equal to, but these details aren't critical here.

Let's generalize our function to take the difference between any two quantiles. We can still consider the max and min, if we like, but we're not limited to that.

19.5 Get something that works, again

The eventual inputs to our new function will be the data `x` and two probabilities.

First, play around with the `quantile()` function. Convince yourself you know how to use it, for example, by cross-checking your results with other built-in functions.

```
quantile(gapminder$lifeExp)
#>   0%  25%  50%  75% 100%
#> 23.6 48.2 60.7 70.8 82.6
quantile(gapminder$lifeExp, probs = 0.5)
#>  50%
#> 60.7
median(gapminder$lifeExp)
#> [1] 60.7
quantile(gapminder$lifeExp, probs = c(0.25, 0.75))
#>  25%  75%
#> 48.2 70.8
boxplot(gapminder$lifeExp, plot = FALSE)$stats
#>      [,1]
#> [1,] 23.6
#> [2,] 48.2
#> [3,] 60.7
#> [4,] 70.8
#> [5,] 82.6
```

Now write a code snippet that takes the difference between two quantiles.

```
the_probs <- c(0.25, 0.75)
the_quantiles <- quantile(gapminder$lifeExp, probs = the_probs)
max(the_quantiles) - min(the_quantiles)
#> [1] 22.6
```

19.6 Turn the working interactive code into a function, again

I'll use `qdiff` as the base of our function's name. I copy the overall structure from our previous "max minus min" work but replace the guts of the function with the more general code we just developed.

```
qdiff1 <- function(x, probs) {
  stopifnot(is.numeric(x))
  the_quantiles <- quantile(x = x, probs = probs)
  max(the_quantiles) - min(the_quantiles)
}
qdiff1(gapminder$lifeExp, probs = c(0.25, 0.75))
#> [1] 22.6
IQR(gapminder$lifeExp) # hey, we've reinvented IQR
#> [1] 22.6
qdiff1(gapminder$lifeExp, probs = c(0, 1))
#> [1] 59
mmm(gapminder$lifeExp)
#> [1] 59
```

Again we do some informal tests against familiar results and external implementations.

19.7 Argument names: freedom and conventions

I want you to understand the importance of argument names.

I can name my arguments almost anything I like. Proof:

```
qdiff2 <- function(zeus, hera) {
  stopifnot(is.numeric(zeus))
  the_quantiles <- quantile(x = zeus, probs = hera)
```

```
  max(the_quantiles) - min(the_quantiles)
}
qdiff2(zeus = gapminder$lifeExp, hera = 0:1)
#> [1] 59
```

While I can name my arguments after Greek gods, it's usually a bad idea. Take all opportunities to make things more self-explanatory via meaningful names.

If you are going to pass the arguments of your function as arguments of a built-in function, consider copying the argument names. Unless you have a good reason to do your own thing (some argument names are bad!), be consistent with the existing function. Again, the reason is to reduce your cognitive load. This is what I've been doing all along and now you know why:

```
qdiff1
#> function(x, probs) {
#>   stopifnot(is.numeric(x))
#>   the_quantiles <- quantile(x = x, probs = probs)
#>   max(the_quantiles) - min(the_quantiles)
#> }
#> <bytecode: 0x7f931961d0a0>
```

We took this detour so you could see there is no *structural* relationship between my arguments (`x` and `probs`) and those of `quantile()` (also `x` and `probs`). The similarity or equivalence of the names **accomplishes nothing** as far as R is concerned; it is solely for the benefit of humans reading, writing, and using the code. Which is very important!

19.8 What a function returns

By this point, I expect someone will have asked about the last line in my function's body. Look above for a reminder of the function's definition.

By default, a function returns the result of the last line of the body. I am just letting that happen with the line `max(the_quantiles) - min(the_quantiles)`. However, there is an explicit function for this: `return()`. I could just as easily make this the last line of my function's body:

```
return(max(the_quantiles) - min(the_quantiles))
```

You absolutely must use `return()` if you want to return early based on some condition, i.e. before execution gets to the last line of the body. Otherwise, you can decide your own conventions about when you use `return()` and when you don't.

19.9 Default values: freedom to NOT specify the arguments

What happens if we call our function but neglect to specify the probabilities?

```
qdiff1(gapminder$lifeExp)
#> Error in quantile(x = x, probs = probs): argument "probs" is missing, with no default
```

Oops! At the moment, this causes a fatal error. It can be nice to provide some reasonable default values for certain arguments. In our case, it would be crazy to specify a default value for the primary input `x`, but very kind to specify a default for `probs`.

We started by focusing on the max and the min, so I think those make reasonable defaults. Here's how to specify that in a function definition.

```
qdiff3 <- function(x, probs = c(0, 1)) {
  stopifnot(is.numeric(x))
  the_quantiles <- quantile(x, probs)
  max(the_quantiles) - min(the_quantiles)
}
```

Again we check how the function works, in old examples and new, specifying the `probs` argument and not.

```
qdiff3(gapminder$lifeExp)
#> [1] 59
mmm(gapminder$lifeExp)
#> [1] 59
qdiff3(gapminder$lifeExp, c(0.1, 0.9))
#> [1] 33.6
```

19.10 Check the validity of arguments, again

Exercise: upgrade our argument validity checks in light of the new argument `probs`.

```
## problems identified during class
## we're not checking that probs is numeric
## we're not checking that probs is length 2
## we're not checking that probs are in [0,1]
```

19.11 Wrap-up and what's next?

Here's the function we've written so far:

```
qdiff3
#> function(x, probs = c(0, 1)) {
#>   stopifnot(is.numeric(x))
#>   the_quantiles <- quantile(x, probs)
#>   max(the_quantiles) - min(the_quantiles)
#> }
#> <bytecode: 0x7f931a5e57b0>
```

What we've accomplished:

- We've generalized our first function to take a difference between arbitrary quantiles.
- We've specified default values for the probabilities that set the quantiles.

Where to next? In part 3 we tackle `NA`s, the special `...` argument, and formal unit testing.

19.12 Resources

- Hadley Wickham's book, Advanced R (2015):
  - Section on function arguments
  - Section on return values

Bibliography

Grolemund, G. and Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software, Articles*, 40(3):1–25.

Spector, P. (2008). *Data Manipulation with R*. Use R! Springer.

White, E. P., Baldridge, E., Brym, Z. T., Locey, K. J., McGlinn, D. J., and Supp, S. R. (2013). Nine simple ways to make it easier to (re)use your data. *PeerJ PrePrints*, 1:e7v2.

Wickham, H. (2011). The split-apply-combine strategy for data analysis. *Journal of Statistical Software, Articles*, 40(1):1–29.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software, Articles*, 59(10):1–23.

Wickham, H. (2015). *Advanced R*. Chapman & Hall/CRC The R Series. CRC Press.

Wickham, H. and Grolemund, G. (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media.