Introduction to Data Science for Transportation Researchers, Planners, and Engineers

Liming Wang

2017-03-15

# Syllabus

Did you ever feel you are “drinking from a hose” with the amount of data you are attempting to analyze? Have you been frustrated with the tedious steps in your data processing and analysis process and thinking there gotta be a better way to do things? Are you curious what the buzz of data science is about? If any of your answers is yes, then this course is for you. Although computing is now an integral part of every aspect of science and engineering, transportation research included, most students of science, engineering, and planning are never taught how to build, use, validate, and share software well. As a result, many spend hours or days doing things badly that could be done well in just a few minutes. The goal of this course is to start changing that so that the students can spend less time wrestling with software and more time doing useful research. Building on the successful data science training programs, such as the Software Carpentry (<http://www.software-carpentry.org/>) and Data Carpentry, and recent development of related software and research, this course exposes students in transportation research and practice to the best practices in scientific computing through hands-on lab sessions and aims to help students tackle the challenge of "drinking from a hose" when dealing with overwhelming amount of data that is increasingly common in transportation research and practice.

## Prerequisite

Basic knowledge and experience of conduct scientific research with quantitative information; skill of using (or keen to learn) a programming language and/or data processing and statistical software (such as python, R, SPSS, Stata).

## Format

Classes will all be hands-on sessions with lecture, discussions and labs. Readings drawn from books, articles, and online resources will be assigned. Students are expected to read them before class and to participate in class discussions. A major component of the class is the class project in which students go through the process of data retrieval, processing, conducting analysis, and developing a report/article while learning the best practices of data science.

## Software and Hardware

This course will use R free statistical software and RStudio (<https://www.rstudio.com/>) as our main interface to R. The lecture and lab instructions will be provided using R. It is possible (and encouraged) for existing Python users (and potentially other software, such as SPSS, Stata, Matlab, SAS, etc) to keep using the software they already know well. Student are encouraged bring their own laptop. The instructor and TA will help the students set up their laptop to run all examples/exercises. They can review/re-run the examples in lectures and labs by themselves.

## Textbook and Readings

The course will use the following textbook:

* [Wickham, H., Grolemund, G., 2017. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data, 1 edition. ed. O’Reilly Media](https://www.amazon.com/Data-Science-Transform-Visualize-Model/dp/1491910399).

An electronic version is available on [Hadley Wickham's website](http://r4ds.had.co.nz/).

For Python users, Wes McKinney’s book is recommended:

* [McKinney, W., 2012. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, 1 edition. ed. O’Reilly Media](https://www.amazon.com/Python-Data-Analysis-Wrangling-IPython/dp/1449319793).

Journal articles and online resources are used as supplements to the textbook.

## Topics

1. Data science best practices
2. Data science work flow
3. Reproducible research

## License

The course materials are made available under the [Creative Commons Attribution license](https://creativecommons.org/licenses/by/4.0/).

## Acknowledgements

This course is developed with the financial support from National Institute of Transportation and Communities project #854.

[](http://nitc.trec.pdx.edu/)

The course materials have greatly benefited from the following sources:

* [R for Data Science](https://github.com/hadley/r4ds) by Hadley Wickham
* [Software Carpentry workshop lessons](https://software-carpentry.org/lessons/)
* [UBC Stat 545](http://stat545.com) by Professor Jenny Bryan at UBC
* [NEU 5110 Introduction to Data Science](http://janvitek.org/events/NEU/5110/) by Professor Jan Vitek

The writing-up and website is powered by the [bookdown](https://bookdown.org) package and [github](https://www.github.com).

# Introduction

## Readings

1. Davenport, T.H., Patil, D.J., 2012. Data Scientist: The Sexiest Job of the 21st Century. Harvard Business Review. URL <https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century> (accessed 2/10/2017).
2. [W] Chapter 1, 6, 8
3. Wilson, G., Aruliah, D.A., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., Haddock, S.H.D., Huff, K., Mitchell, I.M., Plumbley, M., Waugh, B., White, E.P., Wilson, P., 2012. Best Practices for Scientific Computing. arXiv:1210.0530. URL <https://arxiv.org/abs/1210.0530> (accessed 3/7/17).

## Introduction

### Best practices for data science (Wilson et al, 2012)

1. Write programs for people, not computers.
   1. a program should not require its readers to hold more than a handful of facts in memory at once
   2. make names consistent, distinctive, and meaningful
   3. make code style and formatting consistent
2. Let the computer do the work
   1. make the computer repeat tasks
   2. save recent commands in a file for re-use
   3. use a build tool to automate workflows
3. Make incremental changes
   1. work in small steps with frequent feedback and course correction
   2. use a version control system
   3. put everything that has been created manually in version control
4. Don’t repeat yourself (or others)
   1. every piece of data must have a sin- gle authoritative representation in the system
   2. modular- ize code rather than copying and pasting
   3. re-use code instead of rewriting it
5. Plan for mistakes
   1. add assertions to programs to check their operation
   2. use an off-the-shelf unit testing library
   3. turn bugs into test cases
   4. use a symbolic debugger
6. Optimize software only after it works correctly
   1. use a profiler to identify bottlenecks
   2. write code in the highest-level language possible
7. Document design and purpose, not mechan- ics.
   1. document interfaces and reasons, not imple- mentations
   2. refactor code in preference to explaining how it works
   3. embed the documentation for a piece of software in that software
8. Collaborate
   1. use pre-merge code reviews
   2. use pair programming when bring- ing someone new up to speed and when tackling particularly tricky problems
   3. use an issue tracking tool

### The tidyverse work flow

Data Science is a discipline that combines computing with statistics. A data analysis problem is solved in a series of data-centric steps: data acquisition and representation (Import), data cleaning (Tidy), and an iterative sequence of data transformation (Transform), data modelling (Model) and data visualization (Visualize). The end result of the process is to communicate insights obtained from the data (Communicate). This class will take you through all the steps in the process and will teach you how to approach such problems in a systematic manner. You will learn how to design data analysis pipelines as well as how to implement data analysis pipelines in R. The class will also emphasize how elegant code leads to reproducible science.

1. Import your data into R
2. Tidy it
3. Understand your data by iteratively

* visualizing
* tranforming and
* modeling your data

1. Communicate your results to an audience

Program and automate your analysis for easy reuse the whole way through, since you do each of these things on a computer. For this course, we focus on the tidyverse packages namely:

* ggplot2, for data visualisation.
* dplyr, for data manipulation.
* tidyr, for data tidying.
* readr, for data import.
* purrr, for functional programming.
* tibble, for tibbles, a modern re-imagining of data frames.

For more R packages you can use for the tidyverse workflow see [list here](https://github.com/rstudio/RStartHere)



Figure tidyverse workfow

## R and RStudio

### Installation

* Install [R, a free software environment for statistical computing and graphics](http://www.r-project.org) from [CRAN](https://cran.r-project.org), the Comprehensive R Archive Network. Download a precompiled binary distribution for your operating system at <https://cran.r-project.org/> and follow the prompt to install.
* Install RStudio's IDE (stands for *integrated development environment*), a powerful user interface for R. Get the Open Source Edition of RStudio Desktop from <https://www.rstudio.com/products/rstudio/download/>.

Even though it is possible to directly work with the bare R, RStudio provides a much friendlier interface to R and we thus primarily use RStudio as our main interface to R. Since RStudio depends on R, you can not install RStudio without installing R first.

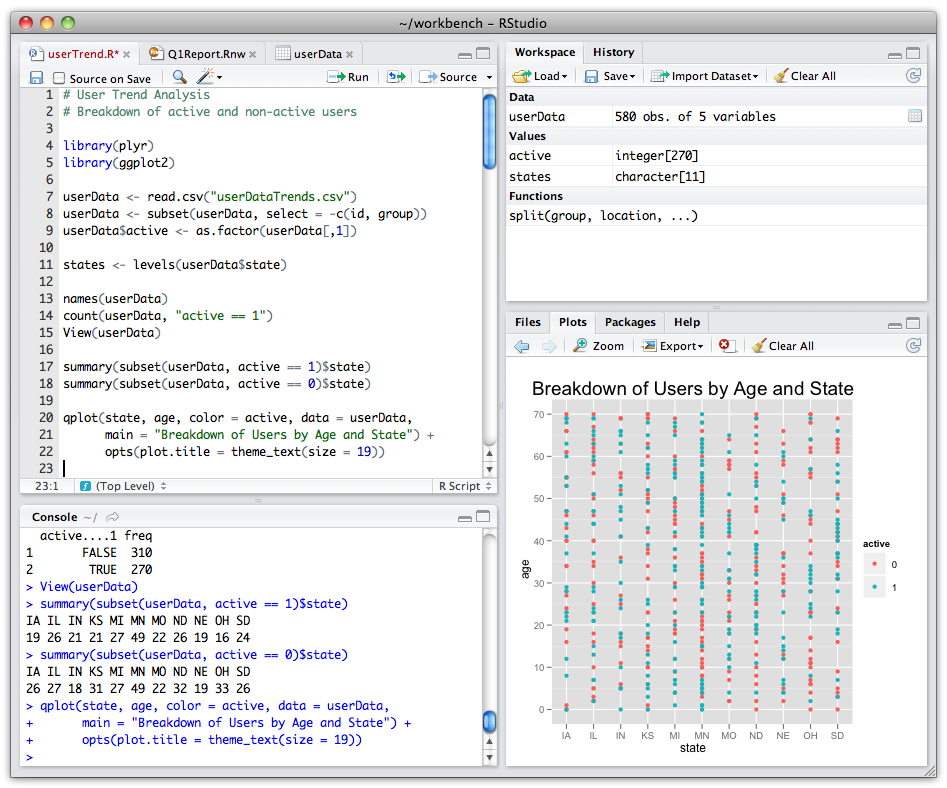
If you have a pre-existing installation of R and/or RStudio, we **highly recommend** that you reinstall both and get as current as possible. It can be considerably harder to run old software than new.

* 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)

### Testing testing

* Do whatever is appropriate for your OS to launch RStudio. You should get a window similar to this screenshot

.

* 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 (followed by enter or return):

x <- 2 \* 4

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.

x

## [1] 8

### Add-on packages

R is an extensible system and many people share useful code they have developed as a *package* via CRAN. To install a package from CRAN, for example the [dplyr](https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html) package for data manipulation, here is one way to do it in the R console (there are others, for example, you can use the *Packages* tab of RStudio to install/update R packages).

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 meta package tidyverse we will cover in this course:

install.packages("tidyverse", dependencies = TRUE)

Nowadays a lot of R package developers share their R package via github (Hadley Wickham being of one of them through <https://github.com/hadley/>). You can install R packages from GitHub with:

install.packages("devtools")  
devtools::install\_github("richfitz/remake")   
#remake is a workflow automation tools that is not (yet) available on CRAN

### R and RStudio tutorial

The RStudio interface is divided into panes for Editing, Console, Environment/History and Misc. You will learn to work with it throughout this course. If you feel you need a tutorial to get familar with RStudio, you can find one at [Introduction to R and RStudio](https://www.sitepoint.com/introduction-r-rstudio/).

### Project orgnization in RStudio

R experts keep all the files associated with a project together — input data, R scripts, analytical results, figures. This is such a wise and common practice that RStudio has built-in support for this via projects. For example, a project can be a R package under development, or code, data and documents for a research project, or a book project (for example, Hadley use a RStudio project for the R for Data Science book at <https://github.com/hadley/r4ds>).

A RStudio project must contain a file with extension Rproj and can contain any files within the directory where the project file lives or its subdirectories. Typical types of files in a RStudio project include: - R Script - R Markdown, R notebook - R Presentation - Shiny Web App - Text file, C++, html, ...

Hadley recommends the following for RStudio projects workflow:

* Create an RStudio project for each data analyis project.
* Keep data files there and load them into R with data import.
* Keep scripts there; edit them, run them in bits or as a whole.
* Save your outputs (plots and cleaned data) there.
* Only ever use relative paths (relative to the root path of the project), not absolute paths.

Everything you need is in one place, and cleanly separated from all the other projects that you are working on.

I recommend organizing files in subdirectories by types, typically with at least these items: - project\_name.Rproj, RStudio project file - README.md, a description file of the project - code, for R scripts - data, for data files - docs, for document/report files, such as R Markdown files

### Set up github and clone datascience course repository

## Learning more

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

* RStudio Cheat Sheet
  + <https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf>
* RStudio's leads for getting help with R
  + <https://support.rstudio.com/hc/en-us/articles/200717153-Getting-Help-with-R>
* R FAQ:
  + <http://cran.r-project.org/doc/FAQ/R-FAQ.html>
* R Installation and Administration
  + <http://cran.r-project.org/doc/manuals/R-admin.html>
* More about add-on packages in the R Installation and Administration Manual
  + <https://cran.r-project.org/doc/manuals/R-admin.html#Add_002don-packages>

# Programming with R

This section has been adapted from Software Carpentry lesson [Programming with R](http://swcarpentry.github.io/r-novice-inflammation/).

## Readings

1. [W] Chapter 4
2. Software Carpentry [Programming with R](http://swcarpentry.github.io/r-novice-inflammation/).

## Starting with Data

We are studying inflammation in patients who have been given a new treatment for arthritis, and need to analyze the first dozen data sets.

The data sets are stored in comma-separated values (CSV) format. Each row holds the observations for just one patient. Each column holds the inflammation measured in a day, so we have a set of values in successive days.

The first few rows of our first file look like this:

## 0,0,1,3,1,2,4,7,8,3,3,3,10,5,7,4,7,7,12,18,6,13,11,11,7,7,4,6,8,8,4,4,5,7,3,4,2,3,0,0  
## 0,1,2,1,2,1,3,2,2,6,10,11,5,9,4,4,7,16,8,6,18,4,12,5,12,7,11,5,11,3,3,5,4,4,5,5,1,1,0,1  
## 0,1,1,3,3,2,6,2,5,9,5,7,4,5,4,15,5,11,9,10,19,14,12,17,7,12,11,7,4,2,10,5,4,2,2,3,2,2,1,1  
## 0,0,2,0,4,2,2,1,6,7,10,7,9,13,8,8,15,10,10,7,17,4,4,7,6,15,6,4,9,11,3,5,6,3,3,4,2,3,2,1  
## 0,1,1,3,3,1,3,5,2,4,4,7,6,5,3,10,8,10,6,17,9,14,9,7,13,9,12,6,7,7,9,6,3,2,2,4,2,0,1,1

We want to:

* Load data into memory,
* Calculate the average value of inflammation per day across all patients, and
* Plot the results.

To do all that, we'll have to learn a little bit about programming.

### Loading Data

To load our inflammation data, first we need to tell our computer where is the file that contains the values. We have been told its name is inflammation-01.csv in the data folder of our project. This is very important in R, if we forget this step we’ll get an error message when trying to read the file. By default, RStudio sets the current working directory to the root of the project, which we can verify with getwd:

getwd()

Just like in the Unix Shell, we type the command and then press Enter (or return).

We can refer to files in our project using path to relative to the project root. (Alternatively you can change the working directory, using the RStudio GUI using the menu option Session -> Set Working Directory -> Choose Directory... or using setwd, but this is not recommended as this uses absolute path and the scripts you create may not work on a different computer.)

The data files are located in the data subdirectory inside the project directory. Now we can load the data into R using read.csv:

require(tidyverse)  
read.csv(file = "data/inflammation-01.csv", header = FALSE)

The expression read.csv(...) is a [function call](%7B%7B%20page.root%20%7D%7D/reference/#function-call) that asks R to run the function read.csv.

read.csv has a few [arguments](%7B%7B%20page.root%20%7D%7D/reference/#argument) include: the name of the file we want to read, and whether the first line of the file contains names for the columns of data.

The filename needs to be a character string (or [string](%7B%7B%20page.root%20%7D%7D/reference/#string) for short), so we put it in quotes. Assigning the second argument, header, to be FALSE indicates that the data file does not have column headers. We'll talk more about the value FALSE, and its converse TRUE later in this class.

**Other Options for Reading CSV Files**

read.csv actually has many more arguments that you may find useful when importing your own data in the future. You can learn more about these options in this supplementary [lesson](%7B%7B%20page.root%20%7D%7D/11-supp-read-write-csv/).

**Loading Data with Headers**

What happens if you forget to put header = FALSE? The default value is header = TRUE, which you can check with ?read.csv or help(read.csv). What do you expect will happen if you leave the default value? Before you run any code, think about what will happen to the first few rows of your data frame, and its overall size. Then run the following code and see if your expectations agree:

read.csv(file = "data/inflammation-01.csv")

The utility of a function is that it will perform its given action on whatever value is passed to the named argument(s). For example, in this case if we provided the name of a different file to the argument file, read.csv would read it instead. We'll learn more of the details about functions and their arguments in the next lesson.

Since we didn't tell it to do anything else with the function's output, the console will display the full contents of the file inflammation-01.csv. Try it out.

read.csv read the file, but we can't use data unless we assign it to a variable. A variable is just a name for a value, such as x, current\_temperature, or subject\_id. We can create a new variable simply by assigning a value to it using <-

weight\_kg <- 55

Once a variable has a value, we can print it by typing the name of the variable and hitting Enter (or return). In general, R will print to the console any object returned by a function or operation *unless* we assign it to a variable.

weight\_kg

## [1] 55

We can do arithmetic with the variable:

# weight in pounds:  
2.2 \* weight\_kg

## [1] 121

**Commenting**

We can add comments to our code using the # character. It is useful to document our code in this way so that others (and us the next time we read it) have an easier time following what the code is doing. {: .callout}

We can also change an object's value by assigning it a new value:

weight\_kg <- 57.5  
# weight in kilograms is now  
weight\_kg

## [1] 57.5

If we imagine the variable as a sticky note with a name written on it, assignment is like putting the sticky note on a particular value:

Variables as Sticky Notes

Variables as Sticky Notes

This means that assigning a value to one object does not change the values of other variables. For example, let's store the subject's weight in pounds in a variable:

weight\_lb <- 2.2 \* weight\_kg  
# weight in kg...  
weight\_kg

## [1] 57.5

# ...and in pounds  
weight\_lb

## [1] 126.5

Creating Another Variable

Creating Another Variable

and then change weight\_kg:

weight\_kg <- 100.0  
# weight in kg now...  
weight\_kg

## [1] 100

# ...and weight in pounds still  
weight\_lb

## [1] 126.5

Updating a Variable

Updating a Variable

Since weight\_lb doesn't "remember" where its value came from, it isn't automatically updated when weight\_kg changes. This is different from the way spreadsheets work.

**Printing with Parentheses**

An alternative way to print the value of a variable is to use () around the assignment statement. As an example: (total\_weight <- weight\_kg + weight\_lb) adds the values of weight\_kg and weight\_lb, assigns the result to the total\_weight, and finally prints the assigned value of the variable total\_weight. {: .callout}

Now that we know how to assign things to variables, let's re-run read.csv and save its result:

dat <- read.csv(file = "data/inflammation-01.csv", header = FALSE)

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value. However, for large data sets it is convenient to use the function head to display only the first few rows of data.

head(dat)

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20  
## 1 0 0 1 3 1 2 4 7 8 3 3 3 10 5 7 4 7 7 12 18  
## 2 0 1 2 1 2 1 3 2 2 6 10 11 5 9 4 4 7 16 8 6  
## 3 0 1 1 3 3 2 6 2 5 9 5 7 4 5 4 15 5 11 9 10  
## 4 0 0 2 0 4 2 2 1 6 7 10 7 9 13 8 8 15 10 10 7  
## 5 0 1 1 3 3 1 3 5 2 4 4 7 6 5 3 10 8 10 6 17  
## 6 0 0 1 2 2 4 2 1 6 4 7 6 6 9 9 15 4 16 18 12  
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38  
## 1 6 13 11 11 7 7 4 6 8 8 4 4 5 7 3 4 2 3  
## 2 18 4 12 5 12 7 11 5 11 3 3 5 4 4 5 5 1 1  
## 3 19 14 12 17 7 12 11 7 4 2 10 5 4 2 2 3 2 2  
## 4 17 4 4 7 6 15 6 4 9 11 3 5 6 3 3 4 2 3  
## 5 9 14 9 7 13 9 12 6 7 7 9 6 3 2 2 4 2 0  
## 6 12 5 18 9 5 3 10 3 12 7 8 4 7 3 5 4 4 3  
## V39 V40  
## 1 0 0  
## 2 0 1  
## 3 1 1  
## 4 2 1  
## 5 1 1  
## 6 2 1

**Assigning Values to Variables**

Draw diagrams showing what variables refer to what values after each statement in the following program:

mass <- 47.5  
age <- 122  
mass <- mass \* 2.0  
age <- age - 20

{: .r} {: .challenge}

### Manipulating Data

Now that our data is loaded in memory, we can start doing things with it. First, let's ask what type of thing dat is:

class(dat)

## [1] "data.frame"

The output tells us that is a data frame. Think of this structure as a spreadsheet in MS Excel that many of us are familiar with. Data frames are very useful for storing data and you will find them elsewhere when programming in R. A typical data frame of experimental data contains individual observations in rows and variables in columns.

We can see the shape, or [dimensions](%7B%7B%20page.root%20%7D%7D/reference/#dimensions), of the data frame with the function dim:

dim(dat)

## [1] 60 40

This tells us that our data frame, dat, has 60 rows and 40 columns.

If we want to get a single value from the data frame, we can provide an [index](%7B%7B%20page.root%20%7D%7D/reference/#index) in square brackets, just as we do in math:

# first value in dat  
dat[1, 1]

## [1] 0

# middle value in dat  
dat[30, 20]

## [1] 16

An index like [30, 20] selects a single element of a data frame, but we can select whole sections as well. For example, we can select the first ten days (columns) of values for the first four patients (rows) like this:

dat[1:4, 1:10]

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10  
## 1 0 0 1 3 1 2 4 7 8 3  
## 2 0 1 2 1 2 1 3 2 2 6  
## 3 0 1 1 3 3 2 6 2 5 9  
## 4 0 0 2 0 4 2 2 1 6 7

The [slice](%7B%7B%20page.root%20%7D%7D/reference/#slice) 1:4 means, "Start at index 1 and go to index 4."

The slice does not need to start at 1, e.g. the line below selects rows 5 through 10:

dat[5:10, 1:10]

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10  
## 5 0 1 1 3 3 1 3 5 2 4  
## 6 0 0 1 2 2 4 2 1 6 4  
## 7 0 0 2 2 4 2 2 5 5 8  
## 8 0 0 1 2 3 1 2 3 5 3  
## 9 0 0 0 3 1 5 6 5 5 8  
## 10 0 1 1 2 1 3 5 3 5 8

We can use the function c, which stands for **c**ombine, to select non-contiguous values:

dat[c(3, 8, 37, 56), c(10, 14, 29)]

## V10 V14 V29  
## 3 9 5 4  
## 8 3 5 6  
## 37 6 9 10  
## 56 7 11 9

We also don't have to provide a slice for either the rows or the columns. If we don't include a slice for the rows, R returns all the rows; if we don't include a slice for the columns, R returns all the columns. If we don't provide a slice for either rows or columns, e.g. dat[, ], R returns the full data frame.

# All columns from row 5  
dat[5, ]

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20  
## 5 0 1 1 3 3 1 3 5 2 4 4 7 6 5 3 10 8 10 6 17  
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38  
## 5 9 14 9 7 13 9 12 6 7 7 9 6 3 2 2 4 2 0  
## V39 V40  
## 5 1 1

# All rows from column 16  
dat[, 16]

## [1] 4 4 15 8 10 15 13 9 11 6 3 8 12 3 5 10 11 4 11 13 15 5 14  
## [24] 13 4 9 13 6 7 6 14 3 15 4 15 11 7 10 15 6 5 6 15 11 15 6  
## [47] 11 15 14 4 10 15 11 6 13 8 4 13 12 9

**Addressing Columns by Name**

Columns can also be addressed by name, with either the $ operator (ie. dat$Age) or square brackets (ie. dat[,'Age']). You can learn more about subsetting by column name in this supplementary [lesson](%7B%7B%20page.root%20%7D%7D/10-supp-addressing-data/). {: .callout}

Now let's perform some common mathematical operations to learn about our inflammation data. When analyzing data we often want to look at partial statistics, such as the maximum value per patient or the average value per day. One way to do this is to select the data we want to create a new temporary data frame, and then perform the calculation on this subset:

# first row, all of the columns  
patient\_1 <- dat[1, ]  
# max inflammation for patient 1  
max(patient\_1)

## [1] 18

**Forcing Conversion**

The code above may give you an error in some R installations, since R does not automatically convert a sliced row of a data.frame to a vector. (Confusingly, sliced columns are automatically converted.) If this happens, you can use the as.numeric command to convert the row of data to a numeric vector:

patient\_1 <- as.numeric(dat[1, ])

max(patient\_1)

You can also check the class of each object:

class(dat[1, ])

class(as.numeric(dat[1, ]))

We don't actually need to store the row in a variable of its own. Instead, we can combine the selection and the function call:

# max inflammation for patient 2  
max(dat[2, ])

## [1] 18

R also has functions for other common calculations, e.g. finding the minimum, mean, median, and standard deviation of the data:

# minimum inflammation on day 7  
min(dat[, 7])

## [1] 1

# mean inflammation on day 7  
mean(dat[, 7])

## [1] 3.8

# median inflammation on day 7  
median(dat[, 7])

## [1] 4

# standard deviation of inflammation on day 7  
sd(dat[, 7])

## [1] 1.725187

What if we need the maximum inflammation for all patients, or the average for each day? As the diagram below shows, we want to perform the operation across a margin of the data frame:

knitr::include\_graphics("fig/r-operations-across-axes.svg")

(#fig:apply_axes)Operations Across Axes.

(#fig:apply\_axes)Operations Across Axes.

To support this, we can use the apply function.

**Getting Help**

To learn about a function in R, e.g. apply, we can read its help documention by running help(apply) or ?apply. {: .callout}

apply allows us to repeat a function on all of the rows (MARGIN = 1) or columns (MARGIN = 2) of a data frame.

Thus, to obtain the average inflammation of each patient we will need to calculate the mean of all of the rows (MARGIN = 1) of the data frame.

avg\_patient\_inflammation <- apply(dat, 1, mean)

And to obtain the average inflammation of each day we will need to calculate the mean of all of the columns (MARGIN = 2) of the data frame.

avg\_day\_inflammation <- apply(dat, 2, mean)

Since the second argument to apply is MARGIN, the above command is equivalent to apply(dat, MARGIN = 2, mean). We'll learn why this is so in the next lesson.

**Efficient Alternatives**

Some common operations have more efficient alternatives. For example, you can calculate the row-wise or column-wise means with rowMeans and colMeans, respectively. {: .callout}

**Slicing (Subsetting) Data**

A subsection of a data frame is called a [slice](%7B%7B%20page.root%20%7D%7D/reference/#slice). We can take slices of character vectors as well:

animal <- c("m", "o", "n", "k", "e", "y")  
# first three characters  
animal[1:3]

## [1] "m" "o" "n"

# last three characters  
animal[4:6]

## [1] "k" "e" "y"

1. If the first four characters are selected using the slice animal[1:4], how can we obtain the first four characters in reverse order?
2. What is animal[-1]? What is animal[-4]? Given those answers, explain what animal[-1:-4] does.
3. Use a slice of animal to create a new character vector that spells the word "eon", i.e. c("e", "o", "n").

**Subsetting More Data**

Suppose you want to determine the maximum inflammation for patient 5 across days three to seven. To do this you would extract the relevant slice from the data frame and calculate the maximum value. Which of the following lines of R code gives the correct answer?

1. max(dat[5, ])
2. max(dat[3:7, 5])
3. max(dat[5, 3:7])
4. max(dat[5, 3, 7])

**Slicing and Re-Assignment**

Using the inflammation data frame dat from above: Let's pretend there was something wrong with the instrument on the first five days for every second patient (#2, 4, 6, etc.), which resulted in the measurements being twice as large as they should be.

1. Write a vector containing each affected patient (hint: ? seq)
2. Create a new data frame with in which you halve the first five days' values in only those patients
3. Print out the corrected data frame to check that your code has fixed the problem

**Using the Apply Function on Patient Data**

Challenge: the apply function can be used to summarize datasets and subsets of data across rows and columns using the MARGIN argument. Suppose you want to calculate the mean inflammation for specific days and patients in the patient dataset (i.e. 60 patients across 40 days).

Please use a combination of the apply function and indexing to:

1. calculate the mean inflammation for patients 1 to 5 over the whole 40 days
2. calculate the mean inflammation for days 1 to 10 (across all patients).
3. calculate the mean inflammation for every second day (across all patients).

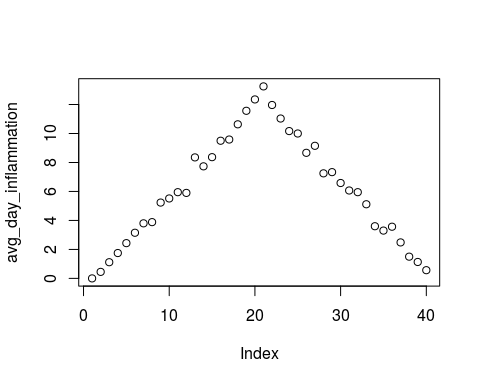
Think about the number of rows and columns you would expect as the result before each apply call and check your intuition by applying the mean function.

### Plotting

The mathematician Richard Hamming once said, "The purpose of computing is insight, not numbers," and the best way to develop insight is often to visualize data. We will come back to visualization later in the course, but let's explore a few of R's plotting features quickly.

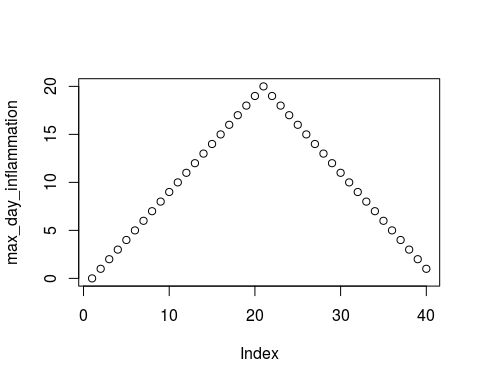
Let's take a look at the average inflammation over time. Recall that we already calculated these values above using apply(dat, 2, mean) and saved them in the variable avg\_day\_inflammation. Plotting the values is done with the function plot.

plot(avg\_day\_inflammation)

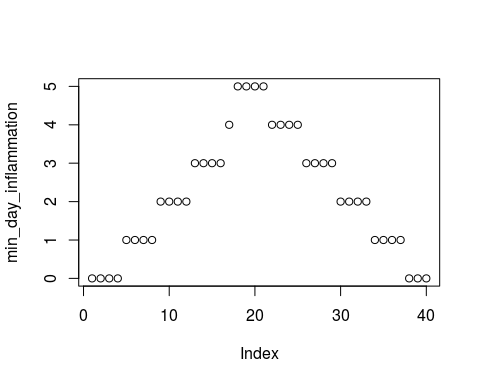


Above, we gave the function plot a vector of numbers corresponding to the average inflammation per day across all patients. plot created a scatter plot where the y-axis is the average inflammation level and the x-axis is the order, or index, of the values in the vector, which in this case correspond to the 40 days of treatment. The result is roughly a linear rise and fall, which is suspicious: based on other studies, we expect a sharper rise and slower fall. Let's have a look at two other statistics: the maximum and minimum inflammation per day.

max\_day\_inflammation <- apply(dat, 2, max)  
plot(max\_day\_inflammation)



min\_day\_inflammation <- apply(dat, 2, min)  
plot(min\_day\_inflammation)



The maximum value rises and falls perfectly smoothly, while the minimum seems to be a step function. Neither result seems particularly likely, so either there's a mistake in our calculations or something is wrong with our data.

**Plotting Data**

Create a plot showing the standard deviation of the inflammation data for each day across all patients.

## Creating Functions

If we only had one data set to analyze, it would probably be faster to load the file into a spreadsheet and use that to plot some simple statistics. But we have twelve files to check, and may have more in the future. In this lesson, we'll learn how to write a function so that we can repeat several operations with a single command.

### Defining a Function

Let's start by defining a function fahr\_to\_kelvin that converts temperatures from Fahrenheit to Kelvin:

fahr\_to\_kelvin <- function(temp) {  
 kelvin <- ((temp - 32) \* (5 / 9)) + 273.15  
 return(kelvin)  
}

We define fahr\_to\_kelvin by assigning it to the output of function. The list of argument names are contained within parentheses. Next, the [body](%7B%7B%20page.root%20%7D%7D/reference/#function-body) of the function--the statements that are executed when it runs--is contained within curly braces ({}). The statements in the body are indented by two spaces, which makes the code easier to read but does not affect how the code operates.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a [return statement](%7B%7B%20page.root%20%7D%7D/reference/#return-statement) to send a result back to whoever asked for it.

**Automatic Returns**

In R, it is not necessary to include the return statement. R automatically returns whichever variable is on the last line of the body of the function. Since we are just learning, we will explicitly define the return statement.

Let's try running our function. Calling our own function is no different from calling any other function:

# freezing point of water  
fahr\_to\_kelvin(32)

## [1] 273.15

# boiling point of water  
fahr\_to\_kelvin(212)

## [1] 373.15

We've successfully called the function that we defined, and we have access to the value that we returned.

### Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

kelvin\_to\_celsius <- function(temp) {  
 celsius <- temp - 273.15  
 return(celsius)  
}  
  
#absolute zero in Celsius  
kelvin\_to\_celsius(0)

## [1] -273.15

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can [compose](%7B%7B%20page.root%20%7D%7D/reference/#function-composition) the two functions we have already created:

fahr\_to\_celsius <- function(temp) {  
 temp\_k <- fahr\_to\_kelvin(temp)  
 result <- kelvin\_to\_celsius(temp\_k)  
 return(result)  
}  
  
# freezing point of water in Celsius  
fahr\_to\_celsius(32.0)

## [1] 0

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-larger chunks to get the effect we want. Real-life functions will usually be larger than the ones shown here--typically half a dozen to a few dozen lines--but they shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on.

**Chaining Functions**

This example showed the output of fahr\_to\_kelvin assigned to temp\_k, which is then passed to kelvin\_to\_celsius to get the final result. It is also possible to perform this calculation in one line of code, by "chaining" functions together, like so:

# freezing point of water in Celsius  
kelvin\_to\_celsius(fahr\_to\_kelvin(32.0))

## [1] 0

**Create a Function**

In the last lesson, we learned to **c**oncatenate elements into a vector using the c function, e.g. x <- c("A", "B", "C") creates a vector x with three elements. Furthermore, we can extend that vector again using c, e.g. y <- c(x, "D") creates a vector y with four elements. Write a function called fence that takes two vectors as arguments, called original and wrapper, and returns a new vector that has the wrapper vector at the beginning and end of the original:

best\_practice <- c("Write", "programs", "for", "people", "not", "computers")  
asterisk <- "\*\*\*" # R interprets a variable with a single value as a vector  
 # with one element.  
fence(best\_practice, asterisk)

## [1] "\*\*\*" "Write" "programs" "for" "people" "not"   
## [7] "computers" "\*\*\*"

If the variable v refers to a vector, then v[1] is the vector's first element and v[length(v)] is its last (the function length returns the number of elements in a vector). Write a function called outside that returns a vector made up of just the first and last elements of its input:

dry\_principle <- c("Don't", "repeat", "yourself", "or", "others")  
outside(dry\_principle)

## [1] "Don't" "others"

**The Call Stack**

For a deeper understanding of how functions work, you'll need to learn how they create their own environments and call other functions. Function calls are managed via the call stack. For more details on the call stack, have a look at the [supplementary material](%7B%7B%20page.root%20%7D%7D/14-supp-call-stack/).

**Named Variables and the Scope of Variables**

* Functions can accept arguments explicitly assigned to a variable name in in the function call functionName(variable = value), as well as arguments by order:

input\_1 = 20  
mySum <- function(input\_1, input\_2 = 10) {  
 output <- input\_1 + input\_2  
 return(output)  
}

1. Given the above code was run, which value does mySum(input\_1 = 1, 3) produce?
   1. 4
   2. 11
   3. 23
   4. 30
2. If mySum(3) returns 13, why does mySum(input\_2 = 3) return an error?

### Testing and Documenting

Once we start putting things in functions so that we can re-use them, we need to start testing that those functions are working correctly. To see how to do this, let's write a function to center a dataset around a particular value:

center <- function(data, desired) {  
 new\_data <- (data - mean(data)) + desired  
 return(new\_data)  
}

We could test this on our actual data, but since we don't know what the values ought to be, it will be hard to tell if the result was correct. Instead, let's create a vector of 0s and then center that around 3. This will make it simple to see if our function is working as expected:

z <- c(0, 0, 0, 0)  
z

## [1] 0 0 0 0

center(z, 3)

## [1] 3 3 3 3

That looks right, so let's try center on our real data. We'll center the inflammation data from day 4 around 0:

dat <- read.csv(file = "data/inflammation-01.csv", header = FALSE)  
centered <- center(dat[, 4], 0)  
head(centered)

## [1] 1.25 -0.75 1.25 -1.75 1.25 0.25

It's hard to tell from the default output whether the result is correct, but there are a few simple tests that will reassure us:

# original min  
min(dat[, 4])

## [1] 0

# original mean  
mean(dat[, 4])

## [1] 1.75

# original max  
max(dat[, 4])

## [1] 3

# centered min  
min(centered)

## [1] -1.75

# centered mean  
mean(centered)

## [1] 0

# centered max  
max(centered)

## [1] 1.25

That seems almost right: the original mean was about 1.75, so the lower bound from zero is now about -1.75. The mean of the centered data is 0. We can even go further and check that the standard deviation hasn't changed:

# original standard deviation  
sd(dat[, 4])

## [1] 1.067628

# centered standard deviation  
sd(centered)

## [1] 1.067628

Those values look the same, but we probably wouldn't notice if they were different in the sixth decimal place. Let's do this instead:

# difference in standard deviations before and after  
sd(dat[, 4]) - sd(centered)

## [1] 0

Sometimes, a very small difference can be detected due to rounding at very low decimal places. R has a useful function for comparing two objects allowing for rounding errors, all.equal:

all.equal(sd(dat[, 4]), sd(centered))

## [1] TRUE

It's still possible that our function is wrong, but it seems unlikely enough that we should probably get back to doing our analysis. We have one more task first, though: we should write some [documentation](%7B%7B%20page.root%20%7D%7D/reference#documentation) for our function to remind ourselves later what it's for and how to use it.

A common way to put documentation in software is to add [comments](%7B%7B%20page.root%20%7D%7D/reference/#comment) like this:

center <- function(data, desired) {  
 # return a new vector containing the original data centered around the  
 # desired value.  
 # Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
 new\_data <- (data - mean(data)) + desired  
 return(new\_data)  
}

**Writing Documentation**

Formal documentation for R functions is written in separate .Rd using a markup language similar to [LaTeX](http://www.latex-project.org/). You see the result of this documentation when you look at the help file for a given function, e.g. ?read.csv. The [roxygen2](http://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html) package allows R coders to write documentation alongside the function code and then process it into the appropriate .Rd files. You will want to switch to this more formal method of writing documentation when you start writing more complicated R projects.

**Functions to Create Graphs**

Write a function called analyze that takes a filename as a argument and displays the three graphs produced in the [previous lesson][01] (average, min and max inflammation over time). analyze("data/inflammation-01.csv") should produce the graphs already shown, while analyze("data/inflammation-02.csv") should produce corresponding graphs for the second data set. Be sure to document your function with comments.

**Rescaling**

Write a function rescale that takes a vector as input and returns a corresponding vector of values scaled to lie in the range 0 to 1. (If and are the lowest and highest values in the original vector, then the replacement for a value should be .) Be sure to document your function with comments.

Test that your rescale function is working properly using min, max, and plot.

### Defining Defaults

We have passed arguments to functions in two ways: directly, as in dim(dat), and by name, as in read.csv(file = "data/inflammation-01.csv", header = FALSE). In fact, we can pass the arguments to read.csv without naming them:

dat <- read.csv("data/inflammation-01.csv", FALSE)

However, the position of the arguments matters if they are not named.

dat <- read.csv(header = FALSE, file = "data/inflammation-01.csv")  
dat <- read.csv(FALSE, "data/inflammation-01.csv")

## Error in read.table(file = file, header = header, sep = sep, quote = quote, : 'file' must be a character string or connection

To understand what's going on, and make our own functions easier to use, let's re-define our center function like this:

center <- function(data, desired = 0) {  
 # return a new vector containing the original data centered around the  
 # desired value (0 by default).  
 # Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
 new\_data <- (data - mean(data)) + desired  
 return(new\_data)  
}

The key change is that the second argument is now written desired = 0 instead of just desired. If we call the function with two arguments, it works as it did before:

test\_data <- c(0, 0, 0, 0)  
center(test\_data, 3)

## [1] 3 3 3 3

But we can also now call center() with just one argument, in which case desired is automatically assigned the default value of 0:

more\_data <- 5 + test\_data  
more\_data

## [1] 5 5 5 5

center(more\_data)

## [1] 0 0 0 0

This is handy: if we usually want a function to work one way, but occasionally need it to do something else, we can allow people to pass an argument when they need to but provide a default to make the normal case easier.

The example below shows how R matches values to arguments

display <- function(a = 1, b = 2, c = 3) {  
 result <- c(a, b, c)  
 names(result) <- c("a", "b", "c") # This names each element of the vector  
 return(result)  
}  
  
# no arguments  
display()

## a b c   
## 1 2 3

# one argument  
display(55)

## a b c   
## 55 2 3

# two arguments  
display(55, 66)

## a b c   
## 55 66 3

# three arguments  
display (55, 66, 77)

## a b c   
## 55 66 77

As this example shows, arguments are matched from left to right, and any that haven't been given a value explicitly get their default value. We can override this behavior by naming the value as we pass it in:

# only setting the value of c  
display(c = 77)

## a b c   
## 1 2 77

**Matching Arguments**

To be precise, R has three ways that arguments are supplied by you are matched to the *formal arguments* of the function definition:

1. by complete name,
2. by partial name (matching on initial *n* characters of the argument name), and
3. by position.

Arguments are matched in the manner outlined above in *that order*: by complete name, then by partial matching of names, and finally by position. {: .callout}

With that in hand, let's look at the help for read.csv():

?read.csv

There's a lot of information there, but the most important part is the first couple of lines:

read.csv(file, header = TRUE, sep = ",", quote = "\"",  
 dec = ".", fill = TRUE, comment.char = "", ...)

This tells us that read.csv() has one argument, file, that doesn't have a default value, and six others that do. Now we understand why the following gives an error:

dat <- read.csv(FALSE, "data/inflammation-01.csv")

## Error in read.table(file = file, header = header, sep = sep, quote = quote, : 'file' must be a character string or connection

It fails because FALSE is assigned to file and the filename is assigned to the argument header.

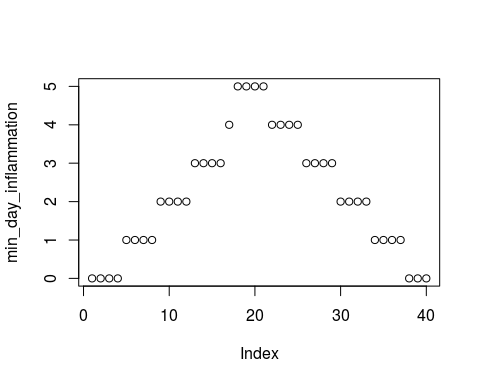
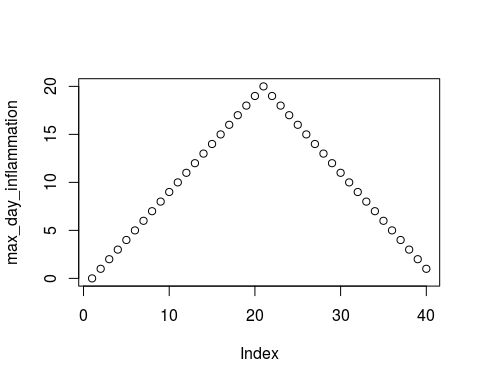
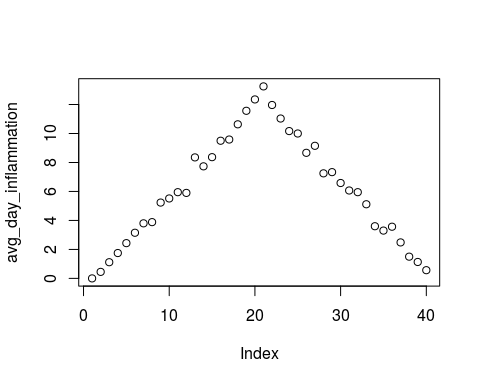
**A Function with Default Argument Values**

Rewrite the rescale function so that it scales a vector to lie between 0 and 1 by default, but will allow the caller to specify lower and upper bounds if they want. Compare your implementation to your neighbor's: do the two functions always behave the same way?

## Loops in R

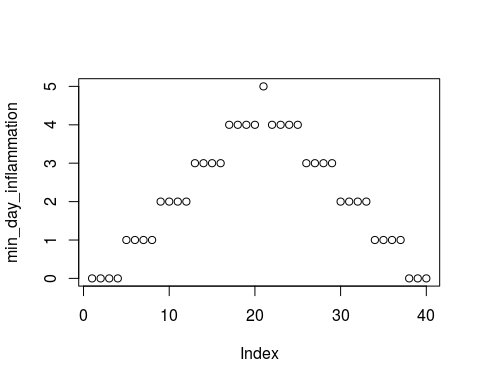
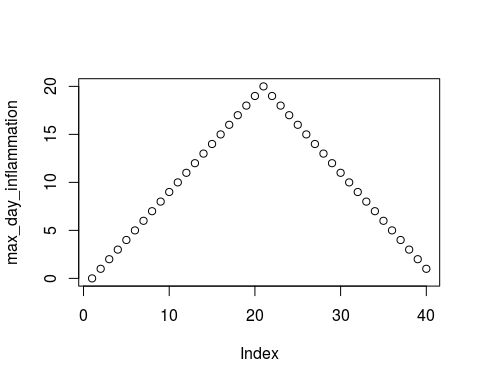
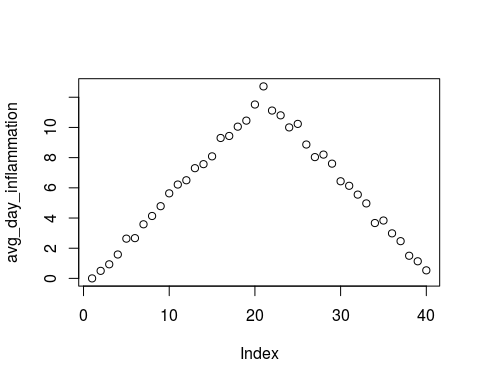
We have created a function called analyze that creates graphs of the minimum, average, and maximum daily inflammation rates for a single data set:

analyze <- function(filename) {  
 # Plots the average, min, and max inflammation over time.  
 # Input is character string of a csv file.  
 dat <- read.csv(file = filename, header = FALSE)  
 avg\_day\_inflammation <- apply(dat, 2, mean)  
 plot(avg\_day\_inflammation)  
 max\_day\_inflammation <- apply(dat, 2, max)  
 plot(max\_day\_inflammation)  
 min\_day\_inflammation <- apply(dat, 2, min)  
 plot(min\_day\_inflammation)  
}  
  
analyze("data/inflammation-01.csv")



We can use it to analyze other data sets one by one:

analyze("data/inflammation-02.csv")



but we have a dozen data sets right now and more on the way. We want to create plots for all our data sets with a single statement. To do that, we'll have to teach the computer how to repeat things.

### For Loops

Suppose we want to print each word in a sentence. One way is to use six print statements:

best\_practice <- c("Let", "the", "computer", "do", "the", "work")  
print\_words <- function(sentence) {  
 print(sentence[1])  
 print(sentence[2])  
 print(sentence[3])  
 print(sentence[4])  
 print(sentence[5])  
 print(sentence[6])  
}  
  
print\_words(best\_practice)

## [1] "Let"  
## [1] "the"  
## [1] "computer"  
## [1] "do"  
## [1] "the"  
## [1] "work"

but that's a bad approach for two reasons:

1. It doesn't scale: if we want to print the elements in a vector that's hundreds long, we'd be better off just typing them in.
2. It's fragile: if we give it a longer vector, it only prints part of the data, and if we give it a shorter input, it returns NA values because we're asking for elements that don't exist!

best\_practice[-6]

## [1] "Let" "the" "computer" "do" "the"

print\_words(best\_practice[-6])

## [1] "Let"  
## [1] "the"  
## [1] "computer"  
## [1] "do"  
## [1] "the"  
## [1] NA

**Not Available**

R has has a special variable, NA, for designating missing values that are **N**ot **A**vailable in a data set. See ?NA and [An Introduction to R](http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Missing-values) for more details. {: .callout}

Here's a better approach:

print\_words <- function(sentence) {  
 for (word in sentence) {  
 print(word)  
 }  
}  
  
print\_words(best\_practice)

## [1] "Let"  
## [1] "the"  
## [1] "computer"  
## [1] "do"  
## [1] "the"  
## [1] "work"

This is shorter---certainly shorter than something that prints every character in a hundred-letter string---and more robust as well:

print\_words(best\_practice[-6])

## [1] "Let"  
## [1] "the"  
## [1] "computer"  
## [1] "do"  
## [1] "the"

The improved version of print\_words uses a [for loop](%7B%7B%20page.root%20%7D%7D/reference/#for-loop) to repeat an operation---in this case, printing---once for each thing in a collection. The general form of a loop is:

for (variable in collection) {  
 do things with variable  
}

We can name the [loop variable](%7B%7B%20page.root%20%7D%7D/reference/#loop-variable) anything we like (with a few [restrictions](http://cran.r-project.org/doc/manuals/R-intro.html#R-commands_003b-case-sensitivity-etc), e.g. the name of the variable cannot start with a digit). in is part of the for syntax. Note that the body of the loop is enclosed in curly braces { }. For a single-line loop body, as here, the braces aren't needed, but it is good practice to include them as we did.

Here's another loop that repeatedly updates a variable:

len <- 0  
vowels <- c("a", "e", "i", "o", "u")  
for (v in vowels) {  
 len <- len + 1  
}  
# Number of vowels  
len

## [1] 5

It's worth tracing the execution of this little program step by step. Since there are five elements in the vector vowels, the statement inside the loop will be executed five times. The first time around, len is zero (the value assigned to it on line 1) and v is "a". The statement adds 1 to the old value of len, producing 1, and updates len to refer to that new value. The next time around, v is "e" and len is 1, so len is updated to be 2. After three more updates, len is 5; since there is nothing left in the vector vowels for R to process, the loop finishes.

Note that a loop variable is just a variable that's being used to record progress in a loop. It still exists after the loop is over, and we can re-use variables previously defined as loop variables as well:

letter <- "z"  
for (letter in c("a", "b", "c")) {  
 print(letter)  
}

## [1] "a"  
## [1] "b"  
## [1] "c"

# after the loop, letter is  
letter

## [1] "c"

Note also that finding the length of a vector is such a common operation that R actually has a built-in function to do it called length:

length(vowels)

## [1] 5

length is much faster than any R function we could write ourselves, and much easier to read than a two-line loop; it will also give us the length of many other things that we haven't met yet, so we should always use it when we can (see this [lesson](%7B%7B%20page.root%20%7D%7D/13-supp-data-structures/) to learn more about the different ways to store data in R).

**Printing Numbers**

R has a built-in function called seq that creates a list of numbers:

seq(3)

## [1] 1 2 3

Using seq, write a function that prints the first **N** natural numbers, one per line:

print\_N(3)

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

**Summing Values**

Write a function called total that calculates the sum of the values in a vector. (R has a built-in function called sum that does this for you. Please don't use it for this exercise.)

ex\_vec <- c(4, 8, 15, 16, 23, 42)  
total(ex\_vec)

## [1] 108

**Exponentiation**

Exponentiation is built into R:

2^4

## [1] 16

Write a function called expo that uses a loop to calculate the same result.

expo(2, 4)

## [1] 16

### Processing Multiple Files

We now have almost everything we need to process all our data files. The only thing that's missing is a function that finds files whose names match a pattern. We do not need to write it ourselves because R already has a function to do this called list.files.

If we run the function without any arguments, list.files(), it returns every file in the current working directory. We can understand this result by reading the help file (?list.files). The first argument, path, is the path to the directory to be searched, and it has the default value of "." (recall from the [lesson](%7B%7B%20site.swc_pages%20%7D%7D/shell-novice/01-filedir/) on the Unix Shell that "." is shorthand for the current working directory). The second argument, pattern, is the pattern being searched, and it has the default value of NULL. Since no pattern is specified to filter the files, all files are returned.

So to list all the csv files, we could run either of the following:

list.files(path = "data", pattern = "csv")

## [1] "car-speeds-cleaned.csv" "car-speeds.csv"   
## [3] "heights.csv" "inflammation-01.csv"   
## [5] "inflammation-02.csv" "inflammation-03.csv"   
## [7] "inflammation-04.csv" "inflammation-05.csv"   
## [9] "inflammation-06.csv" "inflammation-07.csv"   
## [11] "inflammation-08.csv" "inflammation-09.csv"   
## [13] "inflammation-10.csv" "inflammation-11.csv"   
## [15] "inflammation-12.csv" "sample.csv"   
## [17] "small-01.csv" "small-02.csv"   
## [19] "small-03.csv"

list.files(path = "data", pattern = "inflammation")

## [1] "inflammation-01.csv" "inflammation-02.csv" "inflammation-03.csv"  
## [4] "inflammation-04.csv" "inflammation-05.csv" "inflammation-06.csv"  
## [7] "inflammation-07.csv" "inflammation-08.csv" "inflammation-09.csv"  
## [10] "inflammation-10.csv" "inflammation-11.csv" "inflammation-12.csv"

**Organizing Larger Projects**

For larger projects, it is recommended to organize separate parts of the analysis into multiple subdirectories, e.g. one subdirectory for the raw data, one for the code, and one for the results like figures. We have done that here to some extent, putting all of our data files into the subdirectory "data". For more advice on this topic, you can read [A quick guide to organizing computational biology projects](http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.1000424) by William Stafford Noble.

As these examples show, list.files result is a vector of strings, which means we can loop over it to do something with each filename in turn. In our case, the "something" we want is our analyze function.

Because we have put our data in separate subdirectory, if we want to access these files using the output of list.files we also need to include the "path" portion of the file name. We can do that by using the argument full.names = TRUE.

list.files(path = "data", pattern = "csv", full.names = TRUE)

## [1] "data/car-speeds-cleaned.csv" "data/car-speeds.csv"   
## [3] "data/heights.csv" "data/inflammation-01.csv"   
## [5] "data/inflammation-02.csv" "data/inflammation-03.csv"   
## [7] "data/inflammation-04.csv" "data/inflammation-05.csv"   
## [9] "data/inflammation-06.csv" "data/inflammation-07.csv"   
## [11] "data/inflammation-08.csv" "data/inflammation-09.csv"   
## [13] "data/inflammation-10.csv" "data/inflammation-11.csv"   
## [15] "data/inflammation-12.csv" "data/sample.csv"   
## [17] "data/small-01.csv" "data/small-02.csv"   
## [19] "data/small-03.csv"

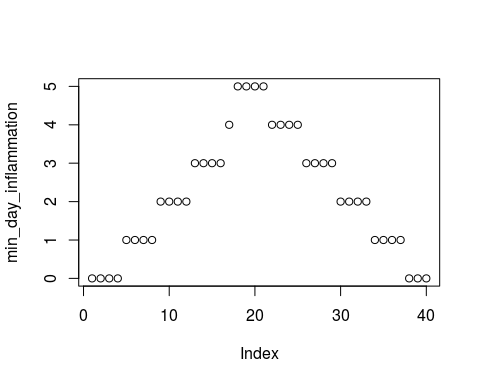
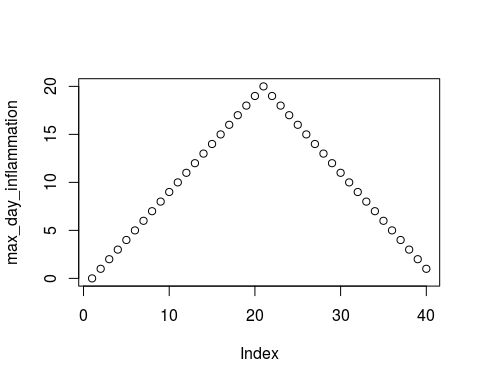
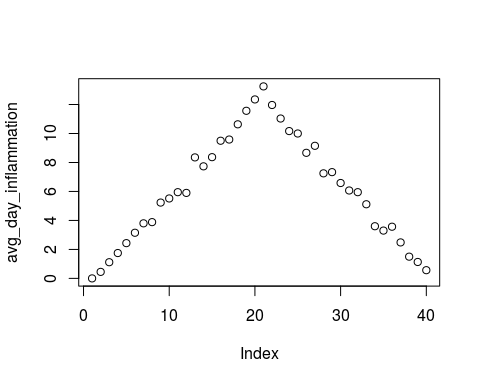
list.files(path = "data", pattern = "inflammation", full.names = TRUE)

## [1] "data/inflammation-01.csv" "data/inflammation-02.csv"  
## [3] "data/inflammation-03.csv" "data/inflammation-04.csv"  
## [5] "data/inflammation-05.csv" "data/inflammation-06.csv"  
## [7] "data/inflammation-07.csv" "data/inflammation-08.csv"  
## [9] "data/inflammation-09.csv" "data/inflammation-10.csv"  
## [11] "data/inflammation-11.csv" "data/inflammation-12.csv"

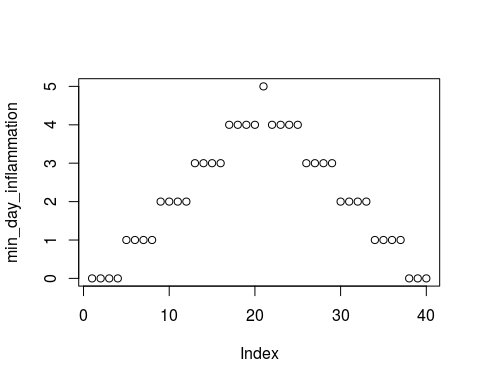
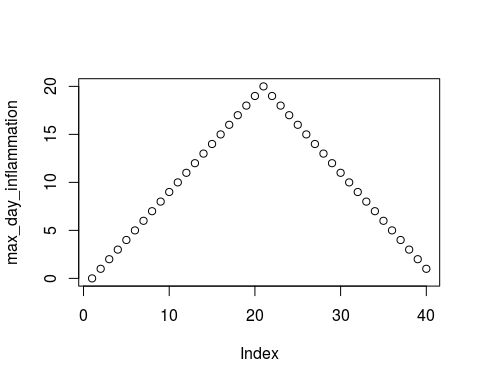
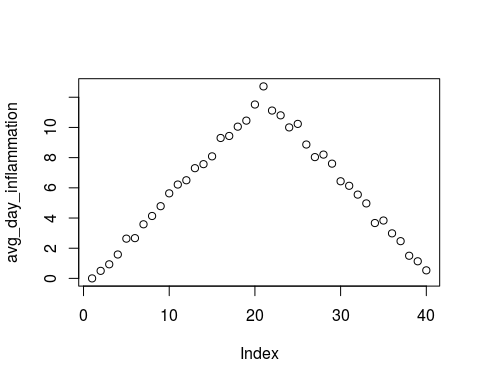
Let's test out running our analyze function by using it on the first three files in the vector returned by list.files:

filenames <- list.files(path = "data", pattern = "inflammation.\*csv", full.names = TRUE)  
filenames <- filenames[1:3]  
for (f in filenames) {  
 print(f)  
 analyze(f)  
}

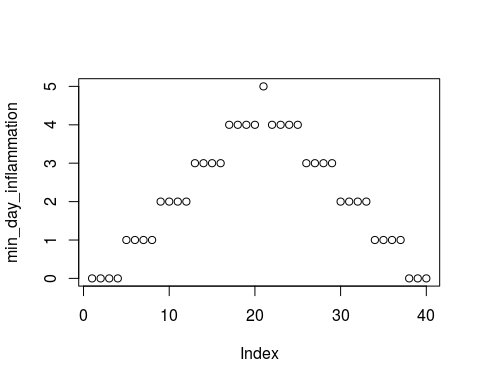
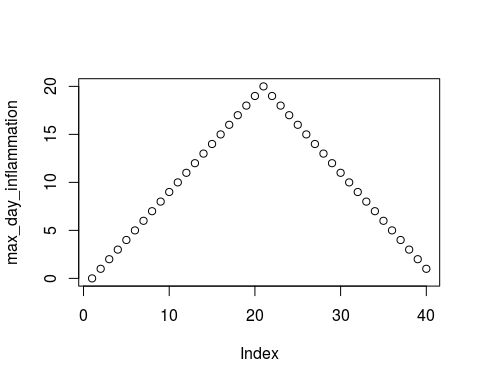
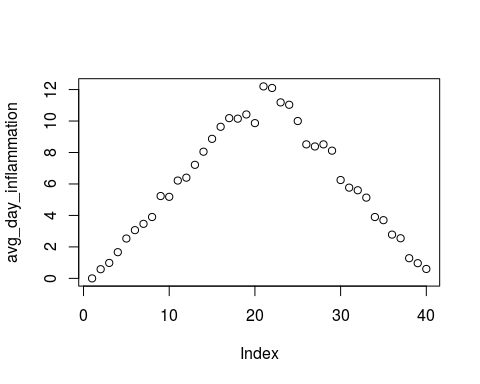
## [1] "data/inflammation-01.csv"



## [1] "data/inflammation-02.csv"



## [1] "data/inflammation-03.csv"



Sure enough, the maxima of these data sets show exactly the same ramp as the first, and their minima show the same staircase structure.

**Other Ways to Do It**

In this lesson we saw how to use a simple for loop to repeat an operation. As you progress with R, you will learn that there are multiple ways to accomplish this. Sometimes the choice of one method over another is more a matter of personal style, but other times it can have consequences for the speed of your code. For instruction on best practices, see [Best Practice for Repetition](#best-practice-for-repetition) below that demonstrates how to properly repeat operations in R.

**Using Loops to Analyze Multiple Files**

Write a function called analyze\_all that takes a filename pattern as its sole argument and runs analyze for each file whose name matches the pattern.

## Best Practice for Repetition

In R you have multiple options when repeating calculations: vectorized operations, for loops, and apply functions.

### Vectorized Operations

A key difference between R and many other languages is a topic known as vectorization. When you wrote the total function, we mentioned that R already has sum to do this; sum is *much* faster than the interpreted for loop because sum is coded in C to work with a vector of numbers. Many of R's functions work this way; the loop is hidden from you in C. Learning to use vectorized operations is a key skill in R.

For example, to add pairs of numbers contained in two vectors

a <- 1:10  
b <- 1:10

You could loop over the pairs adding each in turn, but that would be very inefficient in R.

Instead of using i in a to make our loop variable, we use the function seq\_along to generate indices for each element a contains.

res <- numeric(length = length(a))  
for (i in seq\_along(a)) {  
 res[i] <- a[i] + b[i]  
}  
res

## [1] 2 4 6 8 10 12 14 16 18 20

Instead, + is a *vectorized* function which can operate on entire vectors at once

res2 <- a + b  
all.equal(res, res2)

## [1] TRUE

### Vector Recycling

When performing vector operations in R, it is important to know about recycling. If you perform an operation on two or more vectors of unequal length, R will recycle elements of the shorter vector(s) to match the longest vector. For example:

a <- 1:10  
b <- 1:5  
a + b

## [1] 2 4 6 8 10 7 9 11 13 15

The elements of a and b are added together starting from the first element of both vectors. When R reaches the end of the shorter vector b, it starts again at the first element of b and continues until it reaches the last element of the longest vector a. This behaviour may seem crazy at first glance, but it is very useful when you want to perform the same operation on every element of a vector. For example, say we want to multiply every element of our vector a by 5:

a <- 1:10  
b <- 5  
a \* b

## [1] 5 10 15 20 25 30 35 40 45 50

Remember there are no scalars in R, so b is actually a vector of length 1; in order to add its value to every element of a, it is *recycled* to match the length of a.

When the length of the longer object is a multiple of the shorter object length (as in our example above), the recycling occurs silently. When the longer object length is not a multiple of the shorter object length, a warning is given:

a <- 1:10  
b <- 1:7  
a + b

## [1] 2 4 6 8 10 12 14 9 11 13

### for or apply?

A for loop is used to apply the same function calls to a collection of objects. R has a family of functions, the apply family, which can be used in much the same way. You've already used one of the family, apply in the first [lesson](%7B%7B%20page.root%20%7D%7D/01-starting-with-data/). The apply family members include

* apply - apply over the margins of an array (e.g. the rows or columns of a matrix)
* lapply - apply over an object and return list
* sapply - apply over an object and return a simplified object (an array) if possible
* vapply - similar to sapply but you specify the type of object returned by the iterations

Each of these has an argument FUN which takes a function to apply to each element of the object. Instead of looping over filenames and calling analyze, as you did earlier, you could sapply over filenames with FUN = analyze:

sapply(filenames, FUN = analyze)

Deciding whether to use for or one of the apply family is really personal preference. Using an apply family function forces to you encapsulate your operations as a function rather than separate calls with for. for loops are often more natural in some circumstances; for several related operations, a for loop will avoid you having to pass in a lot of extra arguments to your function.

### Loops in R Are Slow

No, they are not! *If* you follow some golden rules:

1. Don't use a loop when a vectorized alternative exists
2. Don't grow objects (via c, cbind, etc) during the loop - R has to create a new object and copy across the information just to add a new element or row/column
3. Allocate an object to hold the results and fill it in during the loop

As an example, we'll create a new version of analyze that will return the mean inflammation per day (column) of each file.

analyze2 <- function(filenames) {  
 for (f in seq\_along(filenames)) {  
 fdata <- read.csv(filenames[f], header = FALSE)  
 res <- apply(fdata, 2, mean)  
 if (f == 1) {  
 out <- res  
 } else {  
 # The loop is slowed by this call to cbind that grows the object  
 out <- cbind(out, res)  
 }  
 }  
 return(out)  
}  
  
system.time(avg2 <- analyze2(filenames))

## user system elapsed   
## 0.013 0.000 0.011

Note how we add a new column to out at each iteration? This is a cardinal sin of writing a for loop in R.

Instead, we can create an empty matrix with the right dimensions (rows/columns) to hold the results. Then we loop over the files but this time we fill in the fth column of our results matrix out. This time there is no copying/growing for R to deal with.

analyze3 <- function(filenames) {  
 out <- matrix(ncol = length(filenames), nrow = 40) ## assuming 40 here from files  
 for (f in seq\_along(filenames)) {  
 fdata <- read.csv(filenames[f], header = FALSE)  
 out[, f] <- apply(fdata, 2, mean)  
 }  
 return(out)  
}  
  
system.time(avg3 <- analyze3(filenames))

## user system elapsed   
## 0.013 0.000 0.012

In this simple example there is little difference in the compute time of analyze2 and analyze3. This is because we are only iterating over 12 files and hence we only incur 12 copy/grow operations. If we were doing this over more files or the data objects we were growing were larger, the penalty for copying/growing would be much larger.

Note that apply handles these memory allocation issues for you, but then you have to write the loop part as a function to pass to apply. At its heart, apply is just a for loop with extra convenience.

## Make Choices

Our previous lessons have shown us how to manipulate data, define our own functions, and repeat things. However, the programs we have written so far always do the same things, regardless of what data they're given. We want programs to make choices based on the values they are manipulating.

### Saving Plots to a File

So far, we have built a function analyze to plot summary statistics of the inflammation data:

analyze <- function(filename) {  
 # Plots the average, min, and max inflammation over time.  
 # Input is character string of a csv file.  
 dat <- read.csv(file = filename, header = FALSE)  
 avg\_day\_inflammation <- apply(dat, 2, mean)  
 plot(avg\_day\_inflammation)  
 max\_day\_inflammation <- apply(dat, 2, max)  
 plot(max\_day\_inflammation)  
 min\_day\_inflammation <- apply(dat, 2, min)  
 plot(min\_day\_inflammation)  
}

And also built the function analyze\_all to automate the processing of each data file:

analyze\_all <- function(pattern) {  
 # Runs the function analyze for each file in the current working directory  
 # that contains the given pattern.  
 filenames <- list.files(path = "data", pattern = pattern, full.names = TRUE)  
 for (f in filenames) {  
 analyze(f)  
 }  
}

While these are useful in an interactive R session, what if we want to send our results to our collaborators? Since we currently have 12 data sets, running analyze\_all creates 36 plots. Saving each of these individually would be tedious and error-prone. And in the likely situation that we want to change how the data is processed or the look of the plots, we would have to once again save all 36 before sharing the updated results with our collaborators.

Here's how we can save all three plots of the first inflammation data set in a pdf file:

pdf("results/inflammation-01.pdf")  
analyze("data/inflammation-01.csv")  
dev.off()

The function pdf redirects all the plots generated by R into a pdf file, which in this case we have named "inflammation-01.pdf". After we are done generating the plots to be saved in the pdf file, we stop R from redirecting plots with the function dev.off.

**Overwriting Plots**

If you run pdf multiple times without running dev.off, you will save plots to the most recently opened file. However, you won't be able to open the previous pdf files because the connections were not closed. In order to get out of this situation, you'll need to run dev.off until all the pdf connections are closed. You can check your current status using the function dev.cur. If it says "pdf", all your plots are being saved in the last pdf specified. If it says "null device" or "RStudioGD", the plots will be visualized normally.

We can update the analyze function so that it always saves the plots in a pdf. But that would make it more difficult to interactively test out new changes. It would be ideal if analyze would either save or not save the plots based on its input.

### Conditionals

In order to update our function to decide between saving or not, we need to write code that automatically decides between multiple options. The tool R gives us for doing this is called a [conditional statement](%7B%7B%20page.root%20%7D%7D/reference/#conditional-statement), and looks like this:

num <- 37  
if (num > 100) {  
 print("greater")  
} else {  
 print("not greater")  
}  
print("done")

## [1] "not greater"  
## [1] "done"

The second line of this code uses an if statement to tell R that we want to make a choice. If the following test is true, the body of the if (i.e., the lines in the curly braces underneath it) are executed. If the test is false, the body of the else is executed instead. Only one or the other is ever executed:

knitr::include\_graphics("fig/python-flowchart-conditional.svg")

Figure 1 Flowchart Executing a Conditional.

Figure 1 Flowchart Executing a Conditional.

In the example above, the test num > 100 returns the value FALSE, which is why the code inside the if block was skipped and the code inside the else statement was run instead.

num > 100

## [1] FALSE

And as you likely guessed, the opposite of FALSE is TRUE.

num < 100

## [1] TRUE

Conditional statements don't have to include an else. If there isn't one, R simply does nothing if the test is false:

num <- 53  
if (num > 100) {  
 print("num is greater than 100")  
}

We can also chain several tests together when there are more than two options. This makes it simple to write a function that returns the sign of a number:

sign <- function(num) {  
 if (num > 0) {  
 return(1)  
 } else if (num == 0) {  
 return(0)  
 } else {  
 return(-1)  
 }  
}  
  
sign(-3)

## [1] -1

sign(0)

## [1] 0

sign(2/3)

## [1] 1

Note that when combining else and if in an else if statement (similar to elif in Python), the if portion still requires a direct input condition. This is never the case for the else statement alone, which is only executed if all other conditions go unsatisfied. Note that the test for equality uses two equal signs, ==.

**Other Comparisons**

Other tests include greater than or equal to (>=), less than or equal to (<=), and not equal to (!=). {: .callout}

We can also combine tests. An ampersand, &, symbolizes "and". A vertical bar, |, symbolizes "or". & is only true if both parts are true:

if (1 > 0 & -1 > 0) {  
 print("both parts are true")  
} else {  
 print("at least one part is not true")  
}

## [1] "at least one part is not true"

while | is true if either part is true:

if (1 > 0 | -1 > 0) {  
 print("at least one part is true")  
} else {  
 print("neither part is true")  
}

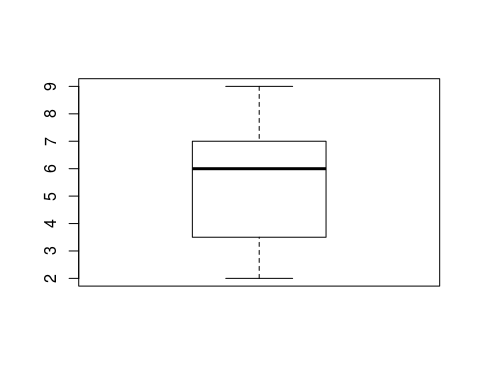
## [1] "at least one part is true"

In this case, "either" means "either or both", not "either one or the other but not both".

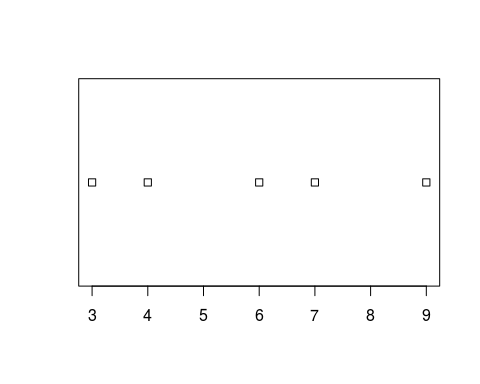
**Choosing Plots Based on Data**

Write a function plot\_dist that plots a boxplot if the length of the vector is greater than a specified threshold and a stripchart otherwise. To do this you'll use the R functions boxplot and stripchart.

dat <- read.csv("data/inflammation-01.csv", header = FALSE)  
plot\_dist(dat[, 10], threshold = 10) # day (column) 10



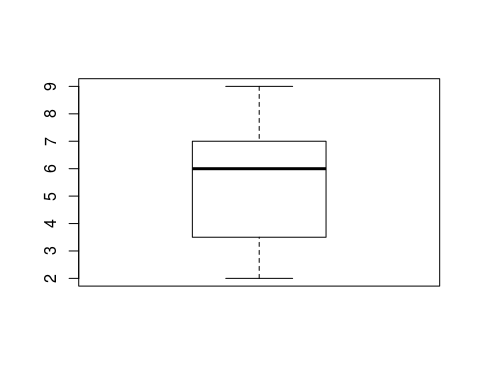
plot\_dist(dat[1:5, 10], threshold = 10) # samples (rows) 1-5 on day (column) 10



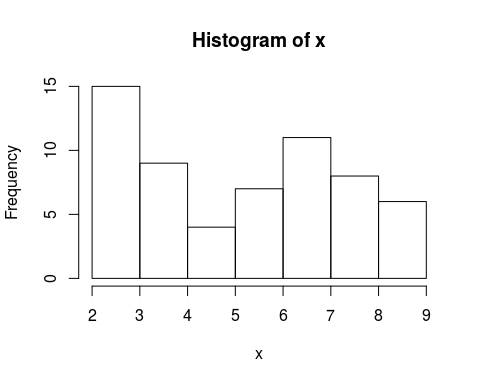
**Histograms Instead**

One of your collaborators prefers to see the distributions of the larger vectors as a histogram instead of as a boxplot. In order to choose between a histogram and a boxplot we will edit the function plot\_dist and add an additional argument use\_boxplot. By default we will set use\_boxplot to TRUE which will create a boxplot when the vector is longer than threshold. When use\_boxplot is set to FALSE, plot\_dist will instead plot a histogram for the larger vectors. As before, if the length of the vector is shorter than threshold, plot\_dist will create a stripchart. A histogram is made with the hist command in R.

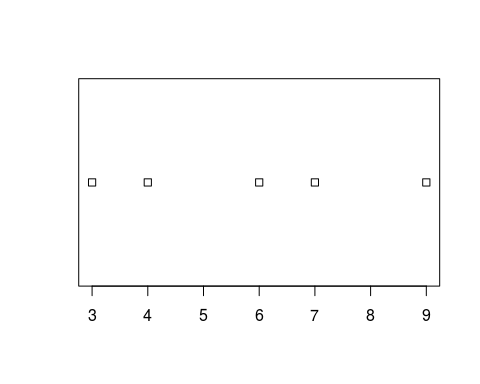
dat <- read.csv("data/inflammation-01.csv", header = FALSE)  
plot\_dist(dat[, 10], threshold = 10, use\_boxplot = TRUE) # day (column) 10 - create boxplot



plot\_dist(dat[, 10], threshold = 10, use\_boxplot = FALSE) # day (column) 10 - create histogram



plot\_dist(dat[1:5, 10], threshold = 10) # samples (rows) 1-5 on day (column) 10



**Find the Maximum Inflammation Score**

Find the file containing the patient with the highest average inflammation score. Print the file name, the patient number (row number) and the value of the maximum average inflammation score.

Tips:

1. Use variables to store the maximum average and update it as you go through files and patients.
2. You can use nested loops (one loop is inside the other) to go through the files as well as through the patients in each file (every row).

Complete the code below:

filenames <- list.files(path = "data", pattern = "inflammation.\*csv", full.names = TRUE)  
filename\_max <- "" # filename where the maximum average inflammation patient is found  
patient\_max <- 0 # index (row number) for this patient in this file  
average\_inf\_max <- 0 # value of the average inflammation score for this patient  
for (f in filenames) {  
 dat <- read.csv(file = f, header = FALSE)  
 dat.means = apply(dat, 1, mean)  
 for (patient\_index in length(dat.means)){  
 patient\_average\_inf = dat.means[patient\_index]  
 # Add your code here ...  
 }  
}  
print(filename\_max)  
print(patient\_max)  
print(average\_inf\_max)

### Saving Automatically Generated Figures

Now that we know how to have R make decisions based on input values, let's update analyze:

analyze <- function(filename, output = NULL) {  
 # Plots the average, min, and max inflammation over time.  
 # Input:  
 # filename: character string of a csv file  
 # output: character string of pdf file for saving  
 if (!is.null(output)) {  
 pdf(output)  
 }  
 dat <- read.csv(file = filename, header = FALSE)  
 avg\_day\_inflammation <- apply(dat, 2, mean)  
 plot(avg\_day\_inflammation)  
 max\_day\_inflammation <- apply(dat, 2, max)  
 plot(max\_day\_inflammation)  
 min\_day\_inflammation <- apply(dat, 2, min)  
 plot(min\_day\_inflammation)  
 if (!is.null(output)) {  
 dev.off()  
 }  
}

We added an argument, output, that by default is set to NULL. An if statement at the beginning checks the argument output to decide whether or not to save the plots to a pdf. Let's break it down. The function is.null returns TRUE if a variable is NULL and FALSE otherwise. The exclamation mark, !, stands for "not". Therefore the line in the if block is only executed if output is "not null".

output <- NULL  
is.null(output)

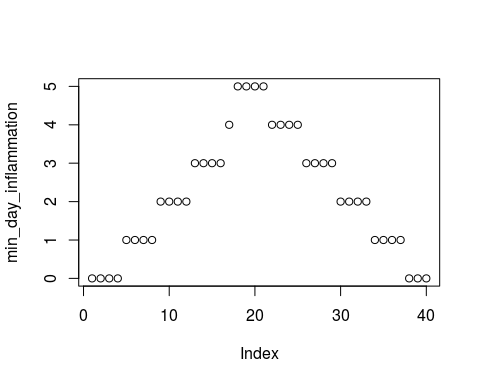
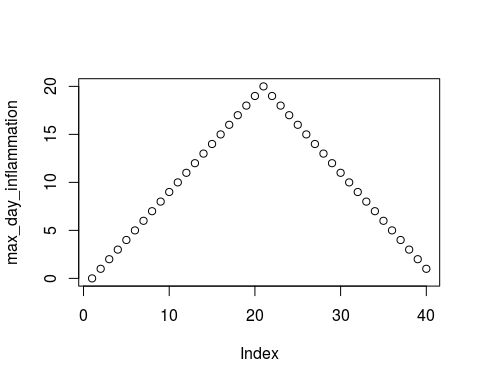
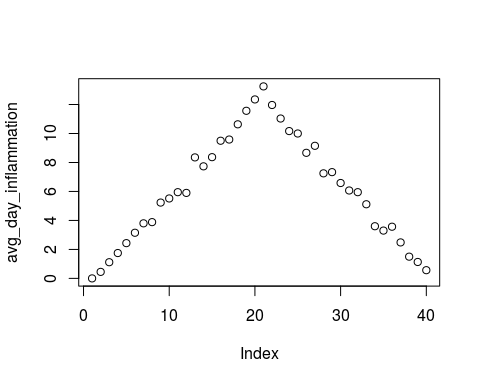
## [1] TRUE

!is.null(output)

## [1] FALSE

Now we can use analyze interactively, as before,

analyze("data/inflammation-01.csv")



but also use it to save plots,

analyze("data/inflammation-01.csv", output = "results/inflammation-01.pdf")

Before going further, we will create a directory results for saving our plots. It is [good practice](http://swcarpentry.github.io/good-enough-practices-in-scientific-computing/) in data analysis projects to save all output to a directory separate from the data and analysis code. You can create this directory using the shell command [mkdir](%7B%7B%20site.swc_pages%20%7D%7D/shell-novice/02-create/), or the R function dir.create()

dir.create("results")

Now run analyze and save the plot in the results directory,

analyze("data/inflammation-01.csv", output = "results/inflammation-01.pdf")

This now works well when we want to process one data file at a time, but how can we specify the output file in analyze\_all? We need to do two things:

1. Substitute the filename ending "csv" with "pdf".
2. Save the plot to the results directory.

To change the extension to "pdf", we will use the function sub,

f <- "inflammation-01.csv"  
sub("csv", "pdf", f)

## [1] "inflammation-01.pdf"

To add the "data" directory to the filename use the function file.path,

file.path("results", sub("csv", "pdf", f))

## [1] "results/inflammation-01.pdf"

Now let's update analyze\_all:

analyze\_all <- function(pattern) {  
 # Directory name containing the data  
 data\_dir <- "data"  
 # Directory name for results  
 results\_dir <- "results"  
 # Runs the function analyze for each file in the current working directory  
 # that contains the given pattern.  
 filenames <- list.files(path = data\_dir, pattern = pattern)  
 for (f in filenames) {  
 pdf\_name <- file.path(results\_dir, sub("csv", "pdf", f))  
 analyze(file.path(data\_dir, f), output = pdf\_name)  
 }  
}

Now we can save all of the results with just one line of code:

analyze\_all("inflammation\*.csv")

Now if we need to make any changes to our analysis, we can edit the analyze function and quickly regenerate all the figures with analyze\_all.

**Changing the Behavior of the Plot Command**

One of your collaborators asks if you can recreate the figures with lines instead of points. Find the relevant argument to plot by reading the documentation (?plot), update analyze, and then recreate all the figures with analyze\_all.

## Command-Line Programs

The R Console and other interactive tools like RStudio are great for prototyping code and exploring data, but sooner or later we will want to use our program in a pipeline or run it in a shell script to process thousands of data files. In order to do that, we need to make our programs work like other Unix command-line tools. For example, we may want a program that reads a data set and prints the average inflammation per patient:

$ Rscript code/readings.R --mean data/inflammation-01.csv  
5.45  
5.425  
6.1  
...  
6.4  
7.05  
5.9

but we might also want to look at the minimum of the first four lines

$ head -4 data/inflammation-01.csv | Rscript code/readings.R --min

or the maximum inflammations in several files one after another:

$ Rscript code/readings.R --max data/inflammation-\*.csv

Our overall requirements are:

1. If no filename is given on the command line, read data from [standard input](%7B%7B%20page.root%20%7D%7D/reference/#standard-input).
2. If one or more filenames are given, read data from them and report statistics for each file separately.
3. Use the --min, --mean, or --max flag to determine what statistic to print.

To make this work, we need to know how to handle command-line arguments in a program, and how to get at standard input. We'll tackle these questions in turn below.

### Command-Line Arguments

Using the text editor of your choice, save the following line of code in a text file called session-info.R:

## sessionInfo()

The function, sessionInfo, outputs the version of R you are running as well as the type of computer you are using (as well as the versions of the packages that have been loaded). This is very useful information to include when asking others for help with your R code.

Now we can run the code in the file we created from the Unix Shell using Rscript:

Rscript code/session-info.R

## R version 3.3.2 (2016-10-31)  
## Platform: x86\_64-pc-linux-gnu (64-bit)  
## Running under: Ubuntu 16.04.1 LTS  
##   
## locale:  
## [1] LC\_CTYPE=en\_US.UTF-8 LC\_NUMERIC=C   
## [3] LC\_TIME=en\_US.UTF-8 LC\_COLLATE=en\_US.UTF-8   
## [5] LC\_MONETARY=en\_US.UTF-8 LC\_MESSAGES=en\_US.UTF-8   
## [7] LC\_PAPER=en\_US.UTF-8 LC\_NAME=C   
## [9] LC\_ADDRESS=C LC\_TELEPHONE=C   
## [11] LC\_MEASUREMENT=en\_US.UTF-8 LC\_IDENTIFICATION=C   
##   
## attached base packages:  
## [1] stats graphics grDevices utils datasets base

**The Right Directory**

If that did not work, you might have seen an error message indicating that the file session-info.R does not exist. Remember that you must be in the correct directory, the one in which you created your script file. You can determine which directory you are currently in using pwd and change to a different directory using cd. For a review, see this [lesson](%7B%7B%20site.swc_pages%20%7D%7D/shell-novice/01-filedir/). {: .callout}

Now let's create another script that does something more interesting. Write the following lines in a file named print-args.R:

## args <- commandArgs()  
## cat(args, sep = "\n")

The function commandArgs extracts all the command line arguments and returns them as a vector. The function cat, similar to the cat of the Unix Shell, outputs the contents of the variable. Since we did not specify a filename for writing, cat sends the output to [standard output](%7B%7B%20page.root%20%7D%7D/reference/#standard-output-(stdout)), which we can then pipe to other Unix functions. Because we set the argument sep to "\n", which is the symbol to start a new line, each element of the vector is printed on its own line. Let's see what happens when we run this program in the Unix Shell:

Rscript code/print-args.R

## /usr/lib/R/bin/exec/R  
## --slave  
## --no-restore  
## --file=code/print-args.R

From this output, we learn that Rscript is just a convenience command for running R scripts. The first argument in the vector is the path to the R executable. The following are all command-line arguments that affect the behavior of R. From the R help file:

* --slave: Make R run as quietly as possible
* --no-restore: Don't restore anything that was created during the R session
* --file: Run this file
* --args: Pass these arguments to the file being run

Thus running a file with Rscript is an easier way to run the following:

R --slave --no-restore --file=code/print-args.R --args

## /usr/lib/R/bin/exec/R  
## --slave  
## --no-restore  
## --file=code/print-args.R  
## --args

If we run it with a few arguments, however:

Rscript code/print-args.R first second third

## /usr/lib/R/bin/exec/R  
## --slave  
## --no-restore  
## --file=code/print-args.R  
## --args  
## first  
## second  
## third

then commandArgs adds each of those arguments to the vector it returns. Since the first elements of the vector are always the same, we can tell commandArgs to only return the arguments that come after --args. Let's update print-args.R and save it as print-args-trailing.R:

## args <- commandArgs(trailingOnly = TRUE)  
## cat(args, sep = "\n")

And then run print-args-trailing from the Unix Shell:

Rscript code/print-args-trailing.R first second third

## first  
## second  
## third

Now commandArgs returns only the arguments that we listed after print-args-trailing.R.

With this in hand, let's build a version of readings.R that always prints the per-patient (per-row) mean of a single data file. The first step is to write a function that outlines our implementation, and a placeholder for the function that does the actual work. By convention this function is usually called main, though we can call it whatever we want. Write the following code in a file called readings-01.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## filename <- args[1]  
## dat <- read.csv(file = filename, header = FALSE)  
## mean\_per\_patient <- apply(dat, 1, mean)  
## cat(mean\_per\_patient, sep = "\n")  
## }

This function gets the name of the file to process from the first element returned by commandArgs. Here's a simple test to run from the Unix Shell:

Rscript code/readings-01.R data/inflammation-01.csv

There is no output because we have defined a function, but haven't actually called it. Let's add a call to main and save it as readings-02.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## filename <- args[1]  
## dat <- read.csv(file = filename, header = FALSE)  
## mean\_per\_patient <- apply(dat, 1, mean)  
## cat(mean\_per\_patient, sep = "\n")  
## }  
##   
## main()

Rscript code/readings-02.R data/inflammation-01.csv

## 5.45  
## 5.425  
## 6.1  
## 5.9  
## 5.55  
## 6.225  
## 5.975  
## 6.65  
## 6.625  
## 6.525  
## 6.775  
## 5.8  
## 6.225  
## 5.75  
## 5.225  
## 6.3  
## 6.55  
## 5.7  
## 5.85  
## 6.55  
## 5.775  
## 5.825  
## 6.175  
## 6.1  
## 5.8  
## 6.425  
## 6.05  
## 6.025  
## 6.175  
## 6.55  
## 6.175  
## 6.35  
## 6.725  
## 6.125  
## 7.075  
## 5.725  
## 5.925  
## 6.15  
## 6.075  
## 5.75  
## 5.975  
## 5.725  
## 6.3  
## 5.9  
## 6.75  
## 5.925  
## 7.225  
## 6.15  
## 5.95  
## 6.275  
## 5.7  
## 6.1  
## 6.825  
## 5.975  
## 6.725  
## 5.7  
## 6.25  
## 6.4  
## 7.05  
## 5.9

**A Simple Command-Line Program**

1. Write a command-line program that does addition and subtraction.

**Hint:** Everything argument read from the command-line is interpreted as a character [string](%7B%7B%20page.root%20%7D%7D/reference/#string). You can convert from a string to a number using the function as.numeric.

Rscript code/arith.R 1 + 2

## 3

Rscript code/arith.R 3 - 4

## -1

1. What goes wrong if you try to add multiplication using \* to the program?
2. Using the function list.files introduced in a previous [lesson](%7B%7B%20page.root%20%7D%7D/03-loops-R/), write a command-line program called find-pattern.R that lists all the files in the current directory that contain a specific pattern:

# For example, searching for the pattern "print-args" returns the two scripts we wrote earlier  
Rscript code/find-pattern.R print-args

### Handling Multiple Files

The next step is to teach our program how to handle multiple files. Since 60 lines of output per file is a lot to page through, we'll start by using three smaller files, each of which has three days of data for two patients. Let's investigate them from the Unix Shell:

ls data/small-\*.csv

## data/small-01.csv  
## data/small-02.csv  
## data/small-03.csv

cat data/small-01.csv

## 0,0,1  
## 0,1,2

Rscript code/readings-02.R data/small-01.csv

## 0.3333333  
## 1

Using small data files as input also allows us to check our results more easily: here, for example, we can see that our program is calculating the mean correctly for each line, whereas we were really taking it on faith before. This is yet another rule of programming: test the simple things first.

We want our program to process each file separately, so we need a loop that executes once for each filename. If we specify the files on the command line, the filenames will be returned by commandArgs(trailingOnly = TRUE). We'll need to handle an unknown number of filenames, since our program could be run for any number of files.

The solution is to loop over the vector returned by commandArgs(trailingOnly = TRUE). Here's our changed program, which we'll save as readings-03.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## for (filename in args) {  
## dat <- read.csv(file = filename, header = FALSE)  
## mean\_per\_patient <- apply(dat, 1, mean)  
## cat(mean\_per\_patient, sep = "\n")  
## }  
## }  
##   
## main()

and here it is in action:

Rscript code/readings-03.R data/small-01.csv data/small-02.csv

## 0.3333333  
## 1  
## 13.66667  
## 11

**Note**: at this point, we have created three versions of our script called readings-01.R, readings-02.R, and readings-03.R. We wouldn't do this in real life: instead, we would have one file called readings.R that we committed to version control every time we got an enhancement working. For teaching, though, we need all the successive versions side by side.

**A Command Line Program with Arguments**

Write a program called check.R that takes the names of one or more inflammation data files as arguments and checks that all the files have the same number of rows and columns. What is the best way to test your program?

### Handling Command-Line Flags

The next step is to teach our program to pay attention to the --min, --mean, and --max flags. These always appear before the names of the files, so let's save the following in readings-04.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## action <- args[1]  
## filenames <- args[-1]  
##   
## for (f in filenames) {  
## dat <- read.csv(file = f, header = FALSE)  
##   
## if (action == "--min") {  
## values <- apply(dat, 1, min)  
## } else if (action == "--mean") {  
## values <- apply(dat, 1, mean)  
## } else if (action == "--max") {  
## values <- apply(dat, 1, max)  
## }  
## cat(values, sep = "\n")  
## }  
## }  
##   
## main()

And we can confirm this works by running it from the Unix Shell:

Rscript code/readings-04.R --max data/small-01.csv

## 1  
## 2

but there are several things wrong with it:

1. main is too large to read comfortably.
2. If action isn't one of the three recognized flags, the program loads each file but does nothing with it (because none of the branches in the conditional match). [Silent failures](%7B%7B%20page.root%20%7D%7D/reference/#silent-failure) like this are always hard to debug.

This version pulls the processing of each file out of the loop into a function of its own. It also checks that action is one of the allowed flags before doing any processing, so that the program fails fast. We'll save it as readings-05.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## action <- args[1]  
## filenames <- args[-1]  
## stopifnot(action %in% c("--min", "--mean", "--max"))  
##   
## for (f in filenames) {  
## process(f, action)  
## }  
## }  
##   
## process <- function(filename, action) {  
## dat <- read.csv(file = filename, header = FALSE)  
##   
## if (action == "--min") {  
## values <- apply(dat, 1, min)  
## } else if (action == "--mean") {  
## values <- apply(dat, 1, mean)  
## } else if (action == "--max") {  
## values <- apply(dat, 1, max)  
## }  
## cat(values, sep = "\n")  
## }  
##   
## main()

This is four lines longer than its predecessor, but broken into more digestible chunks of 8 and 12 lines.

**Parsing Command-Line Flags**

R has a package named [argparse](http://cran.r-project.org/web/packages/argparse/index.html) that helps handle complex command-line flags (it utilizes a [Python module](http://docs.python.org/dev/library/argparse.html) of the same name). We will not cover this package in this lesson but when you start writing programs with multiple parameters you'll want to read through the package's [vignette](http://cran.r-project.org/web/packages/argparse/vignettes/argparse.pdf). {: .callout}

**Shorter Command Line Arguments**

Rewrite this program so that it uses -n, -m, and -x instead of --min, --mean, and --max respectively. Is the code easier to read? Is the program easier to understand?

Separately, modify the program so that if no action is specified (or an incorrect action is given), it prints a message explaining how it should be used.

### Handling Standard Input

The next thing our program has to do is read data from standard input if no filenames are given so that we can put it in a pipeline, redirect input to it, and so on. Let's experiment in another script, which we'll save as count-stdin.R:

## lines <- readLines(con = file("stdin"))  
## count <- length(lines)  
## cat("lines in standard input: ")  
## cat(count, sep = "\n")

This little program reads lines from the program's standard input using file("stdin"). This allows us to do almost anything with it that we could do to a regular file. In this example, we passed it as an argument to the function readLines, which stores each line as an element in a vector. Let's try running it from the Unix Shell as if it were a regular command-line program:

Rscript code/count-stdin.R < data/small-01.csv

## lines in standard input: 2

Note that because we did not specify sep = "\n" when calling cat, the output is written on the same line.

A common mistake is to try to run something that reads from standard input like this:

Rscript code/count-stdin.R data/small-01.csv

i.e., to forget the < character that redirect the file to standard input. In this case, there's nothing in standard input, so the program waits at the start of the loop for someone to type something on the keyboard. We can type some input, but R keeps running because it doesn't know when the standard input has ended. If you ran this, you can pause R by typing ctrl+z (technically it is still paused in the background; if you want to fully kill the process type kill %; see [bash manual](https://www.gnu.org/software/bash/manual/bash.html#Job-Control-Basics) for more information)).

We now need to rewrite the program so that it loads data from file("stdin") if no filenames are provided. Luckily, read.csv can handle either a filename or an open file as its first parameter, so we don't actually need to change process. That leaves main, which we'll update and save as readings-06.R:

## main <- function() {  
## args <- commandArgs(trailingOnly = TRUE)  
## action <- args[1]  
## filenames <- args[-1]  
## stopifnot(action %in% c("--min", "--mean", "--max"))  
##   
## if (length(filenames) == 0) {  
## process(file("stdin"), action)  
## } else {  
## for (f in filenames) {  
## process(f, action)  
## }  
## }  
## }  
##   
## process <- function(filename, action) {  
## dat <- read.csv(file = filename, header = FALSE)  
##   
## if (action == "--min") {  
## values <- apply(dat, 1, min)  
## } else if (action == "--mean") {  
## values <- apply(dat, 1, mean)  
## } else if (action == "--max") {  
## values <- apply(dat, 1, max)  
## }  
## cat(values, sep = "\n")  
## }  
##   
## main()

Let's try it out. Instead of calculating the mean inflammation of every patient, we'll only calculate the mean for the first 10 patients (rows):

head data/inflammation-01.csv | Rscript code/readings-06.R --mean

## 5.45  
## 5.425  
## 6.1  
## 5.9  
## 5.55  
## 6.225  
## 5.975  
## 6.65  
## 6.625  
## 6.525

And now we're done: the program now does everything we set out to do.

## Addressing Data

R is a powerful language for data manipulation. There are three main ways for addressing data inside R objects.

* By index (slicing)
* By logical vector
* By name (columns only)

Lets start by loading some sample data:

dat <- read.csv(file = 'data/sample.csv', header = TRUE, stringsAsFactors = FALSE)

**Interpreting Rows as Headers**

The first row of this csv file is a list of column names. We used the *header=TRUE* argument to read.csv so that R can interpret the file correctly. We are using the *stringsAsFactors=FALSE* argument to override the default behaviour for R. Using factors in R is covered in a separate lesson. {: .callout}

Lets take a look at this data.

class(dat)

## [1] "data.frame"

R has loaded the contents of the .csv file into a variable called dat which is a data frame.

dim(dat)

## [1] 100 9

The data has 100 rows and 9 columns.

head(dat)

## ID Gender Group BloodPressure Age Aneurisms\_q1 Aneurisms\_q2  
## 1 Sub001 m Control 132 16.0 114 140  
## 2 Sub002 m Treatment2 139 17.2 148 209  
## 3 Sub003 m Treatment2 130 19.5 196 251  
## 4 Sub004 f Treatment1 105 15.7 199 140  
## 5 Sub005 m Treatment1 125 19.9 188 120  
## 6 Sub006 M Treatment2 112 14.3 260 266  
## Aneurisms\_q3 Aneurisms\_q4  
## 1 202 237  
## 2 248 248  
## 3 122 177  
## 4 233 220  
## 5 222 228  
## 6 320 294

The data is the results of an (not real) experiment, looking at the number of aneurysms that formed in the eyes of patients who undertook 3 different treatments.

### Addressing by Index

Data can be accessed by index. We have already seen how square brackets [ can be used to subset (slice) data. The generic format is dat[row\_numbers,column\_numbers].

**Selecting Values**

What will be returned by dat[1,1]? {: .challenge}

dat[1,1]

## [1] "Sub001"

If we leave out a dimension R will interpret this as a request for all values in that dimension.

**Selecting More Values**

What will be returned by dat[,2]? {: .challenge}

The colon : can be used to create a sequence of integers.

6:9

## [1] 6 7 8 9

Creates a vector of numbers from 6 to 9.

This can be very useful for addressing data.

**Subsetting with Sequences**

Use the colon operator to index just the aneurism count data (columns 6 to 9). {: .challenge}

Finally we can use the c() (combine) function to address non-sequential rows and columns.

dat[c(1,5,7,9), 1:5]

## ID Gender Group BloodPressure Age  
## 1 Sub001 m Control 132 16.0  
## 5 Sub005 m Treatment1 125 19.9  
## 7 Sub007 f Control 173 17.7  
## 9 Sub009 m Treatment2 131 19.4

Returns the first 5 columns for patients in rows 1,5,7 & 9

**Subsetting Non-Sequential Data**

Return the age and gender values for the first 5 patients. {: .challenge}

### Addressing by Name

Columns in an R data frame are named.

names(dat)

## [1] "ID" "Gender" "Group" "BloodPressure"  
## [5] "Age" "Aneurisms\_q1" "Aneurisms\_q2" "Aneurisms\_q3"   
## [9] "Aneurisms\_q4"

**Default Names**

If names are not specified e.g. using headers=FALSE in a read.csv() function, R assigns default names V1,V2,...,Vn {: .callout}

We usually use the $ operator to address a column by name

dat$Gender

## [1] "m" "m" "m" "f" "m" "M" "f" "m" "m" "f" "m" "f" "f" "m" "m" "m" "f"  
## [18] "m" "m" "F" "f" "m" "f" "f" "m" "M" "M" "f" "m" "f" "f" "m" "m" "m"  
## [35] "m" "f" "f" "m" "M" "m" "f" "m" "m" "m" "f" "f" "M" "M" "m" "m" "m"  
## [52] "f" "f" "f" "m" "f" "m" "m" "m" "f" "f" "f" "f" "M" "f" "m" "f" "f"  
## [69] "M" "m" "m" "m" "F" "m" "m" "f" "M" "M" "M" "f" "m" "M" "M" "m" "m"  
## [86] "f" "f" "f" "m" "m" "f" "m" "F" "f" "m" "m" "F" "m" "M" "M"

Named addressing can also be used in square brackets.

head(dat[,c('Age', 'Gender')])

## Age Gender  
## 1 16.0 m  
## 2 17.2 m  
## 3 19.5 m  
## 4 15.7 f  
## 5 19.9 m  
## 6 14.3 M

**Best Practice**

Best practice is to address columns by name, often you will create or delete columns and the column position will change. {: .callout}

### Logical Indexing

A logical vector contains only the special values TRUE & FALSE.

c(TRUE, TRUE, FALSE, FALSE, TRUE)

## [1] TRUE TRUE FALSE FALSE TRUE

**Truth and Its Opposite**

Note the values TRUE and FALSE are all capital letters and are not quoted. {: .callout}

Logical vectors can be created using relational operators e.g. <, >, ==, !=, %in%.

x <- c(1, 2, 3, 11, 12, 13)  
x < 10

## [1] TRUE TRUE TRUE FALSE FALSE FALSE

x %in% 1:10

## [1] TRUE TRUE TRUE FALSE FALSE FALSE

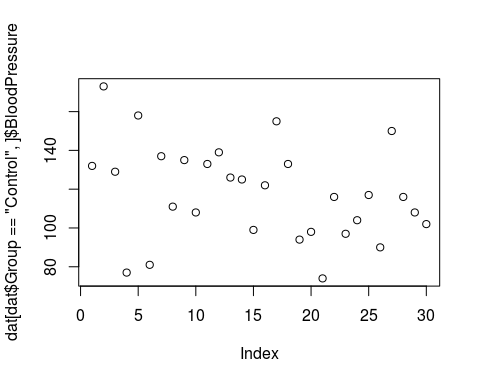
We can use logical vectors to select data from a data frame.

index <- dat$Group == 'Control'  
dat[index,]$BloodPressure

## [1] 132 173 129 77 158 81 137 111 135 108 133 139 126 125 99 122 155  
## [18] 133 94 98 74 116 97 104 117 90 150 116 108 102

Often this operation is written as one line of code:

plot(dat[dat$Group == 'Control',]$BloodPressure)



**Using Logical Indexes**

1. Create a scatterplot showing BloodPressure for subjects not in the control group.
2. How many ways are there to index this set of subjects? {: .challenge}

### Combining Indexing and Assignment

The assignment operator <- can be combined with indexing.

x <- c(1, 2, 3, 11, 12, 13)  
x[x < 10] <- 0  
x

## [1] 0 0 0 11 12 13

**Updating a Subset of Values**

In this dataset, values for Gender have been recorded as both uppercase M, F and lowercase m,f. Combine the indexing and assignment operations to convert all values to lowercase. {: .challenge}

## R Data Structures

### Understanding Basic Data Types in R

To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.

Very important to understand because these are the objects you will manipulate on a day-to-day basis in R. Dealing with object conversions is one of the most common sources of frustration for beginners.

**Everything** in R is an object.

R has 6 (although we will not discuss the raw class for this workshop) atomic vector types.

* character
* numeric (real or decimal)
* integer
* logical
* complex

By *atomic*, we mean the vector only holds data of a single type.

* **character**: "a", "swc"
* **numeric**: 2, 15.5
* **integer**: 2L (the L tells R to store this as an integer)
* **logical**: TRUE, FALSE
* **complex**: 1+4i (complex numbers with real and imaginary parts)

R provides many functions to examine features of vectors and other objects, for example

* class() - what kind of object is it (high-level)?
* typeof() - what is the object's data type (low-level)?
* length() - how long is it? What about two dimensional objects?
* attributes() - does it have any metadata?

# Example  
x <- "dataset"  
typeof(x)

## [1] "character"

attributes(x)

## NULL

y <- 1:10  
y

## [1] 1 2 3 4 5 6 7 8 9 10

typeof(y)

## [1] "integer"

length(y)

## [1] 10

z <- as.numeric(y)  
z

## [1] 1 2 3 4 5 6 7 8 9 10

typeof(z)

## [1] "double"

R has many **data structures**. These include

* atomic vector
* list
* matrix
* data frame
* factors

### Atomic Vectors

A vector is the most common and basic data structure in R and is pretty much the workhorse of R. Technically, vectors can be one of two types:

* atomic vectors
* lists

although the term "vector" most commonly refers to the atomic types not to lists.

### The Different Vector Modes

A vector is a collection of elements that are most commonly of mode character, logical, integer or numeric.

You can create an empty vector with vector(). (By default the mode is logical. You can be more explicit as shown in the examples below.) It is more common to use direct constructors such as character(), numeric(), etc.

vector() # an empty 'logical' (the default) vector

## logical(0)

vector("character", length = 5) # a vector of mode 'character' with 5 elements

## [1] "" "" "" "" ""

character(5) # the same thing, but using the constructor directly

## [1] "" "" "" "" ""

numeric(5) # a numeric vector with 5 elements

## [1] 0 0 0 0 0

logical(5) # a logical vector with 5 elements

## [1] FALSE FALSE FALSE FALSE FALSE

You can also create vectors by directly specifying their content. R will then guess the appropriate mode of storage for the vector. For instance:

x <- c(1, 2, 3)

will create a vector x of mode numeric. These are the most common kind, and are treated as double precision real numbers. If you wanted to explicitly create integers, you need to add an L to each element (or *coerce* to the integer type using as.integer()).

x1 <- c(1L, 2L, 3L)

Using TRUE and FALSE will create a vector of mode logical:

y <- c(TRUE, TRUE, FALSE, FALSE)

While using quoted text will create a vector of mode character:

z <- c("Sarah", "Tracy", "Jon")

### Examining Vectors

The functions typeof(), length(), class() and str() provide useful information about your vectors and R objects in general.

typeof(z)

## [1] "character"

length(z)

## [1] 3

class(z)

## [1] "character"

str(z)

## chr [1:3] "Sarah" "Tracy" "Jon"

**Finding Commonalities**

Do you see a property that's common to all these vectors above? {: .challenge}

### Adding Elements

The function c() (for combine) can also be used to add elements to a vector.

z <- c(z, "Annette")  
z

## [1] "Sarah" "Tracy" "Jon" "Annette"

z <- c("Greg", z)  
z

## [1] "Greg" "Sarah" "Tracy" "Jon" "Annette"

### Vectors from a Sequence of Numbers

You can create vectors as a sequence of numbers.

series <- 1:10  
seq(10)

## [1] 1 2 3 4 5 6 7 8 9 10

seq(from = 1, to = 10, by = 0.1)

## [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3  
## [15] 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7  
## [29] 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1  
## [43] 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6.0 6.1 6.2 6.3 6.4 6.5  
## [57] 6.6 6.7 6.8 6.9 7.0 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9  
## [71] 8.0 8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9 9.0 9.1 9.2 9.3  
## [85] 9.4 9.5 9.6 9.7 9.8 9.9 10.0

### Missing Data

R supports missing data in vectors. They are represented as NA (Not Available) and can be used for all the vector types covered in this lesson:

x <- c(0.5, NA, 0.7)  
x <- c(TRUE, FALSE, NA)  
x <- c("a", NA, "c", "d", "e")  
x <- c(1+5i, 2-3i, NA)

The function is.na() indicates the elements of the vectors that represent missing data, and the function anyNA() returns TRUE if the vector contains any missing values:

x <- c("a", NA, "c", "d", NA)  
y <- c("a", "b", "c", "d", "e")  
is.na(x)

## [1] FALSE TRUE FALSE FALSE TRUE

is.na(y)

## [1] FALSE FALSE FALSE FALSE FALSE

anyNA(x)

## [1] TRUE

anyNA(y)

## [1] FALSE

### Other Special Values

Inf is infinity. You can have either positive or negative infinity.

1/0

## [1] Inf

NaN means Not a Number. It's an undefined value.

0/0

## [1] NaN

### What Happens When You Mix Types Inside a Vector?

R will create a resulting vector with a mode that can most easily accommodate all the elements it contains. This conversion between modes of storage is called "coercion". When R converts the mode of storage based on its content, it is referred to as "implicit coercion". For instance, can you guess what the following do (without running them first)?

xx <- c(1.7, "a")  
xx <- c(TRUE, 2)  
xx <- c("a", TRUE)

You can also control how vectors are coerced explicitly using the as.<class\_name>() functions:

as.numeric("1")

## [1] 1

as.character(1:2)

## [1] "1" "2"

### Objects Attributes

Objects can have **attributes**. Attributes are part of the object. These include:

* names
* dimnames
* dim
* class
* attributes (contain metadata)

You can also glean other attribute-like information such as length (works on vectors and lists) or number of characters (for character strings).

length(1:10)

## [1] 10

nchar("Software Carpentry")

## [1] 18

### Matrix

In R matrices are an extension of the numeric or character vectors. They are not a separate type of object but simply an atomic vector with dimensions; the number of rows and columns.

m <- matrix(nrow = 2, ncol = 2)  
m

## [,1] [,2]  
## [1,] NA NA  
## [2,] NA NA

dim(m)

## [1] 2 2

Matrices in R are filled column-wise.

m <- matrix(1:6, nrow = 2, ncol = 3)

Other ways to construct a matrix

m <- 1:10  
dim(m) <- c(2, 5)

This takes a vector and transforms it into a matrix with 2 rows and 5 columns.

Another way is to bind columns or rows using cbind() and rbind().

x <- 1:3  
y <- 10:12  
cbind(x, y)

## x y  
## [1,] 1 10  
## [2,] 2 11  
## [3,] 3 12

rbind(x, y)

## [,1] [,2] [,3]  
## x 1 2 3  
## y 10 11 12

You can also use the byrow argument to specify how the matrix is filled. From R's own documentation:

mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE)  
mdat

## [,1] [,2] [,3]  
## [1,] 1 2 3  
## [2,] 11 12 13

### List

In R lists act as containers. Unlike atomic vectors, the contents of a list are not restricted to a single mode and can encompass any mixture of data types. Lists are sometimes called generic vectors, because the elements of a list can by of any type of R object, even lists containing further lists. This property makes them fundamentally different from atomic vectors.

A list is a special type of vector. Each element can be a different type.

Create lists using list() or coerce other objects using as.list(). An empty list of the required length can be created using vector()

x <- list(1, "a", TRUE, 1+4i)  
x

## [[1]]  
## [1] 1  
##   
## [[2]]  
## [1] "a"  
##   
## [[3]]  
## [1] TRUE  
##   
## [[4]]  
## [1] 1+4i

x <- vector("list", length = 5) ## empty list  
length(x)

## [1] 5

x[[1]]

## NULL

x <- 1:10  
x <- as.list(x)  
length(x)

## [1] 10

1. What is the class of x[1]?
2. What about x[[1]]?

xlist <- list(a = "Karthik Ram", b = 1:10, data = head(iris))  
xlist

## $a  
## [1] "Karthik Ram"  
##   
## $b  
## [1] 1 2 3 4 5 6 7 8 9 10  
##   
## $data  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 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 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

1. What is the length of this object? What about its structure?

Lists can be extremely useful inside functions. You can “staple” together lots of different kinds of results into a single object that a function can return.

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

Elements are indexed by double brackets. Single brackets will still return a(nother) list.

### Data Frame

A data frame is a very important data type in R. It's pretty much the *de facto* data structure for most tabular data and what we use for statistics.

A data frame is a special type of list where every element of the list has same length.

Data frames can have additional attributes such as rownames(), which can be useful for annotating data, like subject\_id or sample\_id. But most of the time they are not used.

Some additional information on data frames:

* Usually created by read.csv() and read.table().
* Can convert to matrix with data.matrix() (preferred) or as.matrix()
* Coercion will be forced and not always what you expect.
* Can also create with data.frame() function.
* Find the number of rows and columns with nrow(dat) and ncol(dat), respectively.
* Rownames are usually 1, 2, ..., n.

### Creating Data Frames by Hand

To create data frames by hand:

dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)  
dat

## id x y  
## 1 a 1 11  
## 2 b 2 12  
## 3 c 3 13  
## 4 d 4 14  
## 5 e 5 15  
## 6 f 6 16  
## 7 g 7 17  
## 8 h 8 18  
## 9 i 9 19  
## 10 j 10 20

**Useful Data Frame Functions**

* head() - shown first 6 rows
* tail() - show last 6 rows
* dim() - returns the dimensions
* nrow() - number of rows
* ncol() - number of columns
* str() - structure of each column
* names() - shows the names attribute for a data frame, which gives the column names. {: .callout}

See that it is actually a special list:

is.list(iris)

## [1] TRUE

class(iris)

## [1] "data.frame"

|  |  |  |
| --- | --- | --- |
| Dimensions | Homogenous | Heterogeneous |
| 1-D | atomic vector | list |
| 2-D | matrix | data frame |

**Column Types in Data Frames**

Knowing that data frames are lists of lists, can columns be of different type?

What type of structure do you expect on the iris data frame? Hint: Use str().

# The Sepal.Length, Sepal.Width, Petal.Length and Petal.Width columns are all  
# numeric types, while Species is a Factor.  
# Lists can have elements of different types.  
# Since a Data Frame is just a special type of list, it can have columns of  
# differing type (although, remember that type must be consistent within each column!).  
str(iris)

{: .r} {: .challenge}

## Understanding Factors

Factors are used to represent categorical data. Factors can be ordered or unordered and are an important class for statistical analysis and for plotting.

Factors are stored as integers, and have labels associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set values, known as *levels*. By default, R always sorts *levels* in alphabetical order. For instance, if you have a factor with 2 levels:

**The factor() Command**

The factor() command is used to create and modify factors in R:

sex <- factor(c("male", "female", "female", "male"))

{: .callout}

R will assign 1 to the level "female" and 2 to the level "male" (because f comes before m, even though the first element in this vector is "male"). You can check this by using the function levels(), and check the number of levels using nlevels():

levels(sex)

## [1] "female" "male"

nlevels(sex)

## [1] 2

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high") or it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels:

food <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))  
levels(food)

## [1] "high" "low" "medium"

food <- factor(food, levels = c("low", "medium", "high"))  
levels(food)

## [1] "low" "medium" "high"

min(food) ## doesn't work

## Error in Summary.factor(structure(c(1L, 3L, 2L, 3L, 1L, 2L, 3L), .Label = c("low", : 'min' not meaningful for factors

food <- factor(food, levels = c("low", "medium", "high"), ordered=TRUE)  
levels(food)

## [1] "low" "medium" "high"

min(food) ## works!

## [1] low  
## Levels: low < medium < high

In R's memory, these factors are represented by numbers (1, 2, 3). They are better than using simple integer labels because factors are self describing: "low", "medium", and "high"" is more descriptive than 1, 2, 3. Which is low? You wouldn't be able to tell with just integer data. Factors have this information built in. It is particularly helpful when there are many levels (like the subjects in our example data set).

**Representing Data in R**

You have a vector representing levels of exercise undertaken by 5 subjects

**"l","n","n","i","l"** ; n=none, l=light, i=intense

What is the best way to represent this in R?

1. exercise <- c("l", "n", "n", "i", "l")
2. exercise <- factor(c("l", "n", "n", "i", "l"), ordered = TRUE)
3. exercise < -factor(c("l", "n", "n", "i", "l"), levels = c("n", "l", "i"), ordered = FALSE)
4. exercise <- factor(c("l", "n", "n", "i", "l"), levels = c("n", "l", "i"), ordered = TRUE) {: .challenge}

### Converting Factors

Converting from a factor to a number can cause problems:

f <- factor(c(3.4, 1.2, 5))  
as.numeric(f)

## [1] 2 1 3

This does not behave as expected (and there is no warning).

The recommended way is to use the integer vector to index the factor levels:

levels(f)[f]

## [1] "3.4" "1.2" "5"

This returns a character vector, the as.numeric() function is still required to convert the values to the proper type (numeric).

f <- levels(f)[f]  
f <- as.numeric(f)

### Using Factors

Lets load our example data to see the use of factors:

dat <- read.csv(file = 'data/sample.csv', stringsAsFactors = TRUE)

**Default Behavior**

stringsAsFactors=TRUE is the default behavior for R. We could leave this argument out. It is included here for clarity. {: .callout}

str(dat)

## 'data.frame': 100 obs. of 9 variables:  
## $ ID : Factor w/ 100 levels "Sub001","Sub002",..: 1 2 3 4 5 6 7 8 9 10 ...  
## $ Gender : Factor w/ 4 levels "f","F","m","M": 3 3 3 1 3 4 1 3 3 1 ...  
## $ Group : Factor w/ 3 levels "Control","Treatment1",..: 1 3 3 2 2 3 1 3 3 1 ...  
## $ BloodPressure: int 132 139 130 105 125 112 173 108 131 129 ...  
## $ Age : num 16 17.2 19.5 15.7 19.9 14.3 17.7 19.8 19.4 18.8 ...  
## $ Aneurisms\_q1 : int 114 148 196 199 188 260 135 216 117 188 ...  
## $ Aneurisms\_q2 : int 140 209 251 140 120 266 98 238 215 144 ...  
## $ Aneurisms\_q3 : int 202 248 122 233 222 320 154 279 181 192 ...  
## $ Aneurisms\_q4 : int 237 248 177 220 228 294 245 251 272 185 ...

Notice the first 3 columns have been converted to factors. These values were text in the data file so R automatically interpreted them as categorical variables.

summary(dat)

## ID Gender Group BloodPressure Age   
## Sub001 : 1 f:35 Control :30 Min. : 62.0 Min. :12.10   
## Sub002 : 1 F: 4 Treatment1:35 1st Qu.:107.5 1st Qu.:14.78   
## Sub003 : 1 m:46 Treatment2:35 Median :117.5 Median :16.65   
## Sub004 : 1 M:15 Mean :118.6 Mean :16.42   
## Sub005 : 1 3rd Qu.:133.0 3rd Qu.:18.30   
## Sub006 : 1 Max. :173.0 Max. :20.00   
## (Other):94   
## Aneurisms\_q1 Aneurisms\_q2 Aneurisms\_q3 Aneurisms\_q4   
## Min. : 65.0 Min. : 80.0 Min. :105.0 Min. :116.0   
## 1st Qu.:118.0 1st Qu.:131.5 1st Qu.:182.5 1st Qu.:186.8   
## Median :158.0 Median :162.5 Median :217.0 Median :219.0   
## Mean :158.8 Mean :168.0 Mean :219.8 Mean :217.9   
## 3rd Qu.:188.0 3rd Qu.:196.8 3rd Qu.:248.2 3rd Qu.:244.2   
## Max. :260.0 Max. :283.0 Max. :323.0 Max. :315.0   
##

Notice the summary() function handles factors differently to numbers (and strings), the occurrence counts for each value is often more useful information.

**The summary() Function**

The summary() function is a great way of spotting errors in your data (look at the *dat$Gender* column). It's also a great way for spotting missing data. {: .callout}

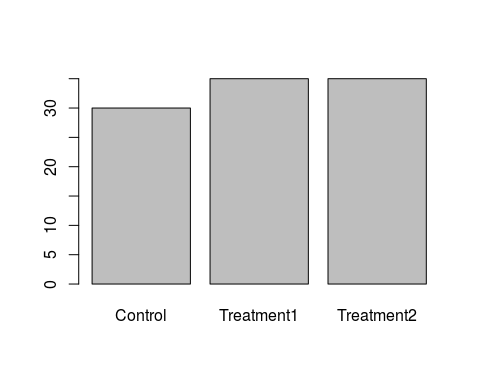
**Reordering Factors**

The function table() tabulates observations and can be used to create bar plots quickly. For instance:

table(dat$Group)

##   
## Control Treatment1 Treatment2   
## 30 35 35

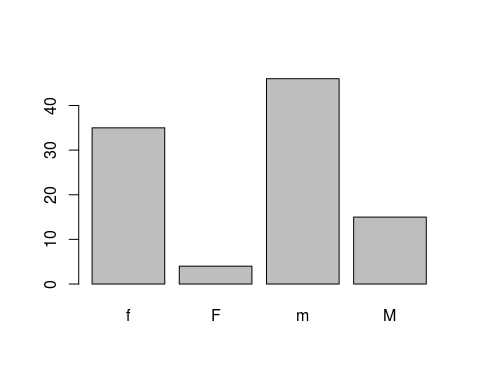
barplot(table(dat$Group))

 Use the factor() command to modify the column dat$Group so that the *control* group is plotted last {: .challenge}

### Removing Levels from a Factor

Some of the Gender values in our dataset have been coded incorrectly. Let's remove factors.

barplot(table(dat$Gender))

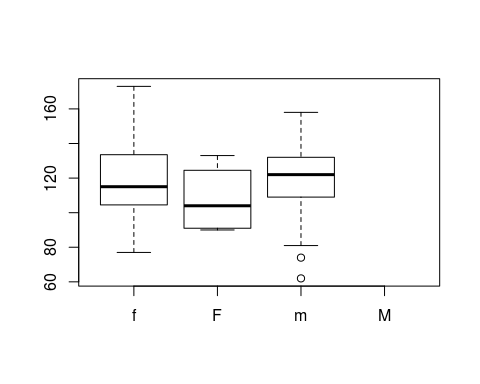


Values should have been recorded as lowercase 'm' & 'f'. We should correct this.

dat$Gender[dat$Gender == 'M'] <- 'm'

**Updating Factors**

plot(x = dat$Gender, y = dat$BloodPressure)

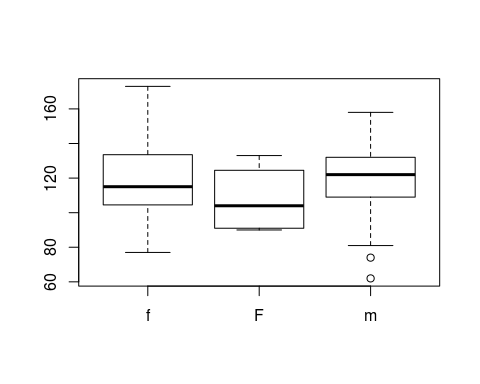


Why does this plot show 4 levels?

*Hint* how many levels does dat$Gender have? {: .challenge}

We need to tell R that "M" is no longer a valid value for this column. We use the droplevels() function to remove extra levels.

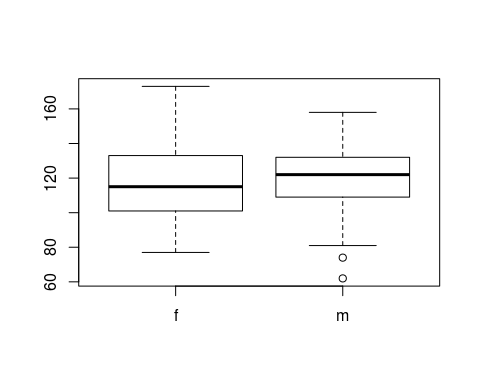
dat$Gender <- droplevels(dat$Gender)  
plot(x = dat$Gender, y = dat$BloodPressure)



**Adjusting Factor Levels**

Adjusting the levels() of a factor provides a useful shortcut for reassigning values in this case.

levels(dat$Gender)[2] <- 'f'  
plot(x = dat$Gender, y = dat$BloodPressure)

 {: .callout}

## Learning more

1. Interactive course [Introduction to R](https://www.datacamp.com/courses/free-introduction-to-r) by Data Camp covers the R basics, vectors, matrices, factors and data frame.
2. Interactive course [Intermediate R](https://www.datacamp.com/courses/free-introduction-to-r) by Data Camp covers Conditionals, Control Flow, loops, functions, and utility functions.
3. [Google's R Style Guide](https://google.github.io/styleguide/Rguide.xml) lists Google's coding rules for R to make code easier to read, share, and verify.
4. Hadley Wickham's [Style Guide](http://adv-r.had.co.nz/Style.html) in his Advanced R book.

# Version Control with Git

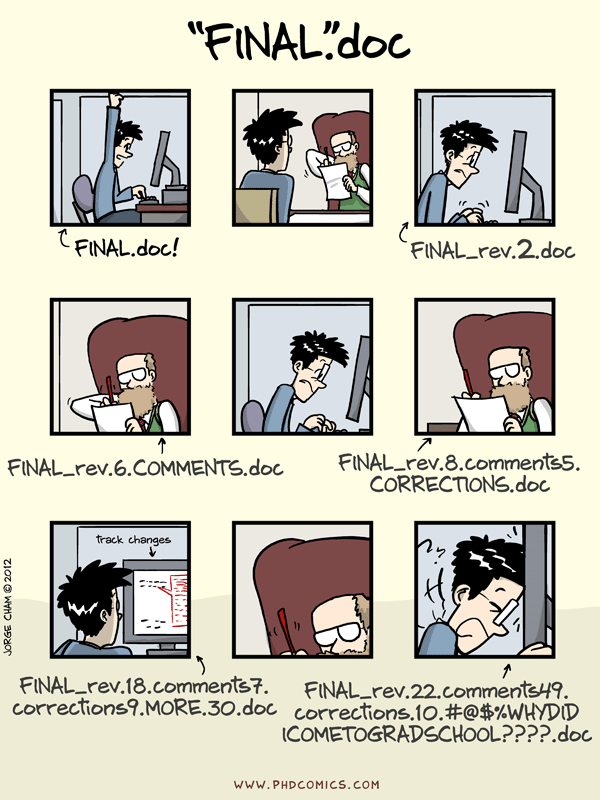
This section has been adapted from Software Carpentry lesson [Version Control with Git](http://swcarpentry.github.io/git-novice/).

## Readings

1. Software Carpentry [Version Control with Git](http://swcarpentry.github.io/git-novice/).
2. Hadley Wickham [Git and Github](http://r-pkgs.had.co.nz/git.html)

## Introduction

We'll start by exploring how version control can be used to keep track of what one person did and when. Even if you aren't collaborating with other people, automated version control is much better than this situation:

[](http://www.phdcomics.com)

"Piled Higher and Deeper" by Jorge Cham, <http://www.phdcomics.com>

We've all been in this situation before: it seems ridiculous to have multiple nearly-identical versions of the same document. Some word processors let us deal with this a little better, such as Microsoft Word's [Track Changes](https://support.office.com/en-us/article/Track-changes-in-Word-197ba630-0f5f-4a8e-9a77-3712475e806a) or Google Docs' [version history](https://support.google.com/docs/answer/190843?hl=en).

Version control systems start with a base version of the document and then save just the changes you made at each step of the way. You can think of it as a tape: if you rewind the tape and start at the base document, then you can play back each change and end up with your latest version.

Changes Are Saved Sequentially

Changes Are Saved Sequentially

Once you think of changes as separate from the document itself, you can then think about "playing back" different sets of changes onto the base document and getting different versions of the document. For example, two users can make independent sets of changes based on the same document.

Different Versions Can be Saved

Different Versions Can be Saved

If there aren't conflicts, you can even play two sets of changes onto the same base document.

Multiple Versions Can be Merged

Multiple Versions Can be Merged

A version control system is a tool that keeps track of these changes for us and helps us version and merge our files. It allows you to decide which changes make up the next version, called a [commit](%7B%7B%20page.root%20%7D%7D/reference/#commit), and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a [repository](%7B%7B%20page.root%20%7D%7D/reference/#repository). Repositories can be kept in sync across different computers facilitating collaboration among different people.

**The Long History of Version Control Systems**

Automated version control systems are nothing new. Tools like RCS, CVS, or Subversion have been around since the early 1980s and are used by many large companies. However, many of these are now becoming considered as legacy systems due to various limitations in their capabilities. In particular, the more modern systems, such as Git and [Mercurial](http://swcarpentry.github.io/hg-novice/) are *distributed*, meaning that they do not need a centralized server to host the repository. These modern systems also include powerful merging tools that make it possible for multiple authors to work within the same files concurrently.

**Paper Writing**

* Imagine you drafted an excellent paragraph for a paper you are writing, but later ruin it. How would you retrieve the *excellent* version of your conclusion? Is it even possible?
* Imagine you have 5 co-authors. How would you manage the changes and comments they make to your paper? If you use LibreOffice Writer or Microsoft Word, what happens if you accept changes made using the Track Changes option? Do you have a history of those changes?

## Setting up Git

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

* our name and email address,
* to colorize our output,
* what our preferred text editor is,
* and that we want to use these settings globally (i.e. for every project)

On a command line, Git commands are written as git verb, where verb is what we actually want to do. So here is how Dracula sets up his new laptop:

$ git config --global user.name "Vlad Dracula"  
$ git config --global user.email "vlad@tran.sylvan.ia"  
$ git config --global color.ui "auto"

Please use your own name and email address instead of Dracula's. This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to [GitHub](http://github.com/), [BitBucket](http://bitbucket.org/), [GitLab](http://gitlab.com/) or another Git host server in a later lesson will include this information. If you are concerned about privacy, please review [GitHub's instructions for keeping your email address private](https://help.github.com/articles/keeping-your-email-address-private/).

He also has to set his favorite text editor, following this table:

|  |  |
| --- | --- |
| Editor | Configuration command |
| Atom | $ git config --global core.editor "atom --wait" |
| nano | $ git config --global core.editor "nano -w" |
| Text Wrangler (Mac) | $ git config --global core.editor "edit -w" |
| Sublime Text (Mac) | $ git config --global core.editor "subl -n -w" |
| Sublime Text (Win, 32-bit install) | $ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime\_text.exe' -w" |
| Sublime Text (Win, 64-bit install) | $ git config --global core.editor "'c:/program files/sublime text 3/sublime\_text.exe' -w" |
| Notepad++ (Win, 32-bit install) | $ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin" |
| Notepad++ (Win, 64-bit install) | $ git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin" |
| Kate (Linux) | $ git config --global core.editor "kate" |
| Gedit (Linux) | $ git config --global core.editor "gedit --wait --new-window" |
| Scratch (Linux) | $ git config --global core.editor "scratch-text-editor" |
| emacs | $ git config --global core.editor "emacs" |
| vim | $ git config --global core.editor "vim" |

It is possible to reconfigure the text editor for Git whenever you want to change it.

**Exiting Vim**

Note that vim is the default editor for for many programs, if you haven't used vim before and wish to exit a session, type Esc then :q! and Enter.

The four commands we just ran above only need to be run once: the flag --global tells Git to use the settings for every project, in your user account, on this computer.

You can check your settings at any time:

$ git config --list

You can change your configuration as many times as you want: just use the same commands to choose another editor or update your email address.

**Proxy**

In some networks you need to use a [proxy](https://en.wikipedia.org/wiki/Proxy_server). If this is the case, you may also need to tell Git about the proxy:

$ git config --global http.proxy proxy-url  
$ git config --global https.proxy proxy-url

To disable the proxy, use

$ git config --global --unset http.proxy  
$ git config --global --unset https.proxy

**Git Help and Manual**

Always remember that if you forget a git command, you can access the list of command by using -h and access the git manual by using --help :

$ git config -h  
$ git config --help

## Creating a Repository

Once Git is configured, we can start using it. Let's create a directory for our work and then move into that directory:

$ mkdir planets  
$ cd planets

Then we tell Git to make planets a [repository](%7B%7B%20page.root%20%7D%7D/reference/#repository)—a place where Git can store versions of our files:

$ git init

If we use ls to show the directory's contents, it appears that nothing has changed:

$ ls

But if we add the -a flag to show everything, we can see that Git has created a hidden directory within planets called .git:

$ ls -a

. .. .git

Git stores information about the project in this special sub-directory. If we ever delete it, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

$ git status

# On branch master  
#  
# Initial commit  
#  
nothing to commit (create/copy files and use "git add" to track)

**Places to Create Git Repositories**

Dracula starts a new project, moons, related to his planets project. Despite Wolfman's concerns, he enters the following sequence of commands to create one Git repository inside another:

$ cd # return to home directory  
$ mkdir planets # make a new directory planets  
$ cd planets # go into planets  
$ git init # make the planets directory a Git repository  
$ mkdir moons # make a sub-directory planets/moons  
$ cd moons # go into planets/moons  
$ git init # make the moons sub-directory a Git repository

Why is it a bad idea to do this? (Notice here that the planets project is now also tracking the entire moons repository.) How can Dracula undo his last git init?

## Tracking Changes

Let's create a file called mars.txt that contains some notes about the Red Planet's suitability as a base. (We'll use nano to edit the file; you can use whatever editor you like. In particular, this does not have to be the core.editor you set globally earlier.)

$ nano mars.txt

Type the text below into the mars.txt file:

Cold and dry, but everything is my favorite color

mars.txt now contains a single line, which we can see by running:

$ ls

mars.txt

$ cat mars.txt

Cold and dry, but everything is my favorite color

If we check the status of our project again, Git tells us that it's noticed the new file:

$ git status

On branch master  
  
Initial commit  
  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
  
 mars.txt  
nothing added to commit but untracked files present (use "git add" to track)

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git to track a file using git add:

$ git add mars.txt

and then check that the right thing happened:

$ git status

On branch master  
  
Initial commit  
  
Changes to be committed:  
 (use "git rm --cached <file>..." to unstage)  
  
 new file: mars.txt

Git now knows that it's supposed to keep track of mars.txt, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

$ git commit -m "Start notes on Mars as a base"

[master (root-commit) f22b25e] Start notes on Mars as a base  
 1 file changed, 1 insertion(+)  
 create mode 100644 mars.txt

When we run git commit, Git takes everything we have told it to save by using git add and stores a copy permanently inside the special .git directory. This permanent copy is called a [commit](%7B%7B%20page.root%20%7D%7D/reference/#commit) (or [revision](%7B%7B%20page.root%20%7D%7D/reference/#revision)) and its short identifier is f22b25e (Your commit may have another identifier.)

We use the -m flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run git commit without the -m option, Git will launch nano (or whatever other editor we configured as core.editor) so that we can write a longer message.

[Good commit messages](http://chris.beams.io/posts/git-commit/) start with a brief (<50 characters) summary of changes made in the commit. If you want to go into more detail, add a blank line between the summary line and your additional notes.

If we run git status now:

$ git status

On branch master  
nothing to commit, working directory clean

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using git log:

$ git log

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 09:51:46 2013 -0400  
  
 Start notes on Mars as a base

git log lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the git commit command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

**Where Are My Changes?**

If we run ls at this point, we will still see just one file called mars.txt. That's because Git saves information about files' history in the special .git directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Now suppose Dracula adds more information to the file. (Again, we'll edit with nano and then cat the file to show its contents; you may use a different editor, and don't need to cat.)

$ nano mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman

When we run git status now, it tells us that a file it already knows about has been modified:

$ git status

On branch master  
Changes not staged for commit:  
 (use "git add <file>..." to update what will be committed)  
 (use "git checkout -- <file>..." to discard changes in working directory)  
  
 modified: mars.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with git add) nor have we saved them (which we do with git commit). So let's do that now. It is good practice to always review our changes before saving them. We do this using git diff. This shows us the differences between the current state of the file and the most recently saved version:

$ git diff

diff --git a/mars.txt b/mars.txt  
index df0654a..315bf3a 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +1,2 @@  
 Cold and dry, but everything is my favorite color  
+The two moons may be a problem for Wolfman

The output is cryptic because it is actually a series of commands for tools like editors and patch telling them how to reconstruct one file given the other. If we break it down into pieces:

1. The first line tells us that Git is producing output similar to the Unix diff command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; df0654a and 315bf3a are unique computer-generated labels for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where we added a line.

After reviewing our change, it's time to commit it:

$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"  
$ git status

On branch master  
Changes not staged for commit:  
 (use "git add <file>..." to update what will be committed)  
 (use "git checkout -- <file>..." to discard changes in working directory)  
  
 modified: mars.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")

Whoops: Git won't commit because we didn't use git add first. Let's fix that:

$ git add mars.txt  
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"

[master 34961b1] Add concerns about effects of Mars' moons on Wolfman  
 1 file changed, 1 insertion(+)

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to our supervisor's work to our thesis. We might want to commit those additions, and the corresponding addition to the bibliography, but *not* commit the work we're doing on the conclusion (which we haven't finished yet).

To allow for this, Git has a special *staging area* where it keeps track of things that have been added to the current [change set](%7B%7B%20page.root%20%7D%7D/reference/#change-set) but not yet committed.

**Staging Area**

If you think of Git as taking snapshots of changes over the life of a project, git add specifies *what* will go in a snapshot (putting things in the staging area), and git commit then *actually takes* the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type git commit, Git will prompt you to use git commit -a or git commit --all, which is kind of like gathering *everyone* for the picture! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to snapshots, you might get the extra with incomplete makeup walking on the stage for the snapshot because you used -a!) Try to stage things manually, or you might find yourself searching for "git undo commit" more than you would like!

The Git Staging Area

The Git Staging Area

Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

$ nano mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity

$ git diff

diff --git a/mars.txt b/mars.txt  
index 315bf3a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1,2 +1,3 @@  
 Cold and dry, but everything is my favorite color  
 The two moons may be a problem for Wolfman  
+But the Mummy will appreciate the lack of humidity

So far, so good: we've added one line to the end of the file (shown with a + in the first column). Now let's put that change in the staging area and see what git diff reports:

$ git add mars.txt  
$ git diff

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

$ git diff --staged

diff --git a/mars.txt b/mars.txt  
index 315bf3a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1,2 +1,3 @@  
 Cold and dry, but everything is my favorite color  
 The two moons may be a problem for Wolfman  
+But the Mummy will appreciate the lack of humidity

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

$ git commit -m "Discuss concerns about Mars' climate for Mummy"

[master 005937f] Discuss concerns about Mars' climate for Mummy  
 1 file changed, 1 insertion(+)

check our status:

$ git status

On branch master  
nothing to commit, working directory clean

and look at the history of what we've done so far:

$ git log

commit 005937fbe2a98fb83f0ade869025dc2636b4dad5  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 10:14:07 2013 -0400  
  
 Discuss concerns about Mars' climate for Mummy  
  
commit 34961b159c27df3b475cfe4415d94a6d1fcd064d  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 10:07:21 2013 -0400  
  
 Add concerns about effects of Mars' moons on Wolfman  
  
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 09:51:46 2013 -0400  
  
 Start notes on Mars as a base

**Word-based diffing**

Sometimes, e.g. in the case of the text documments a line-wise diff is too coarse. That is where the --color-words option of git diff comes in very useful as it highlights the changed words using colors.

**Paging the Log**

When the output of git log is too long to fit in your screen, git uses a program to split it into pages of the size of your screen. When this "pager" is called, you will notice that the last line in your screen is a :, instead of your usual prompt.

* To get out of the pager, press q.
* To move to the next page, press the space bar.
* To search for some\_word in all pages, type /some\_word and navigate throught matches pressing n.

**Limit Log Size**

To avoid having git log cover your entire terminal screen, you can limit the number of commits that Git lists by using -N, where N is the number of commits that you want to view. For example, if you only want information from the last commit you can use:

$ git log -1

commit 005937fbe2a98fb83f0ade869025dc2636b4dad5  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 10:14:07 2013 -0400  
  
 Discuss concerns about Mars' climate for Mummy

You can also reduce the quantity of information using the --oneline option:

$ git log --oneline

\* 005937f Discuss concerns about Mars' climate for Mummy  
\* 34961b1 Add concerns about effects of Mars' moons on Wolfman  
\* f22b25e Start notes on Mars as a base

You can also combine the --oneline options with others. One useful combination is:

$ git log --oneline --graph --all --decorate

\* 005937f Discuss concerns about Mars' climate for Mummy (HEAD, master)  
\* 34961b1 Add concerns about effects of Mars' moons on Wolfman  
\* f22b25e Start notes on Mars as a base

**Directories**

Two important facts you should know about directories in Git.

1. Git does not track directories on their own, only files within them. Try it for yourself:

$ mkdir directory  
$ git status  
$ git add directory  
$ git status

Note, our newly created empty directory directory does not appear in the list of untracked files even if we explicitly add it (*via* git add) to our repository. This is the reason why you will sometimes see .gitkeep files in otherwise empty directories. Unlike .gitignore, these files are not special and their sole purpose is to populate a directory so that Git adds it to the repository. In fact, you can name such files anything you like.

1. If you create a directory in your Git repository and populate it with files, you can add all files in the directory at once by:

git add <directory-with-files>

To recap, when we want to add changes to our repository, we first need to add the changed files to the staging area (git add) and then commit the staged changes to the repository (git commit):

The Git Commit Workflow

The Git Commit Workflow

**Choosing a Commit Message**

Which of the following commit messages would be most appropriate for the last commit made to mars.txt?

1. "Changes"
2. "Added line 'But the Mummy will appreciate the lack of humidity' to mars.txt"
3. "Discuss effects of Mars' climate on the Mummy"

**Committing Changes to Git**

Which command(s) below would save the changes of myfile.txt to my local Git repository?

1. $ git commit -m "my recent changes"
2. $ git init myfile.txt  
   $ git commit -m "my recent changes"
3. $ git add myfile.txt  
   $ git commit -m "my recent changes"
4. $ git commit -m myfile.txt "my recent changes"

**Committing Multiple Files**

The staging area can hold changes from any number of files that you want to commit as a single snapshot.

1. Add some text to mars.txt noting your decision to consider Venus as a base
2. Create a new file venus.txt with your initial thoughts about Venus as a base for you and your friends
3. Add changes from both files to the staging area, and commit those changes.

**Author and Committer**

For each of the commits you have done, Git stored your name twice. You are named as the author and as the committer. You can observe that by telling Git to show you more information about your last commits:

$ git log --format=full

When commiting you can name someone else as the author:

$ git commit --author="Vlad Dracula <vlad@tran.sylvan.ia>"

Create a new repository and create two commits: one without the --author option and one by naming a colleague of yours as the author. Run git log and git log --format=full. Think about ways how that can allow you to collaborate with your colleagues.

## Exploring History

As we saw in the previous lesson, we can refer to commits by their identifiers. You can refer to the *most recent commit* of the working directory by using the identifier HEAD.

We've been adding one line at a time to mars.txt, so it's easy to track our progress by looking, so let's do that using our HEADs. Before we start, let's make a change to mars.txt.

$ nano mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
An ill-considered change

Now, let's see what we get.

$ git diff HEAD mars.txt

diff --git a/mars.txt b/mars.txt  
index b36abfd..0848c8d 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1,3 +1,4 @@  
 Cold and dry, but everything is my favorite color  
 The two moons may be a problem for Wolfman  
 But the Mummy will appreciate the lack of humidity  
+An ill-considered change.

which is the same as what you would get if you leave out HEAD (try it). The real goodness in all this is when you can refer to previous commits. We do that by adding ~1 to refer to the commit one before HEAD.

$ git diff HEAD~1 mars.txt

If we want to see the differences between older commits we can use git diff again, but with the notation HEAD~1, HEAD~2, and so on, to refer to them:

$ git diff HEAD~2 mars.txt

diff --git a/mars.txt b/mars.txt  
index df0654a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +1,3 @@  
 Cold and dry, but everything is my favorite color  
+The two moons may be a problem for Wolfman  
+But the Mummy will appreciate the lack of humidity

We could also use git show which shows us what changes we made at an older commit as well as the commit message, rather than the *differences* between a commit and our working directory that we see by using git diff.

$ git show HEAD~2 mars.txt

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d  
Author: Vlad Dracula <vlad@tran.sylvan.ia>  
Date: Thu Aug 22 10:07:21 2013 -0400  
  
 Add concerns about effects of Mars' moons on Wolfman  
  
diff --git a/mars.txt b/mars.txt  
index df0654a..315bf3a 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +1,2 @@  
 Cold and dry, but everything is my favorite color  
+The two moons may be a problem for Wolfman

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as HEAD; we can refer to previous commits using the ~ notation, so HEAD~1 (pronounced "head minus one") means "the previous commit", while HEAD~123 goes back 123 commits from where we are now.

We can also refer to commits using those long strings of digits and letters that git log displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID f22b25e3233b4645dabd0d81e651fe074bd8e73b, so let's try this:

$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt

diff --git a/mars.txt b/mars.txt  
index df0654a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +1,3 @@  
 Cold and dry, but everything is my favorite color  
+The two moons may be a problem for Wolfman  
+But the Mummy will appreciate the lack of humidity

That's the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters:

$ git diff f22b25e mars.txt

diff --git a/mars.txt b/mars.txt  
index df0654a..b36abfd 100644  
--- a/mars.txt  
+++ b/mars.txt  
@@ -1 +1,3 @@  
 Cold and dry, but everything is my favorite color  
+The two moons may be a problem for Wolfman  
+But the Mummy will appreciate the lack of humidity

All right! So we can save changes to files and see what we've changed—now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

$ nano mars.txt  
$ cat mars.txt

We will need to manufacture our own oxygen

git status now tells us that the file has been changed, but those changes haven't been staged:

$ git status

On branch master  
Changes not staged for commit:  
 (use "git add <file>..." to update what will be committed)  
 (use "git checkout -- <file>..." to discard changes in working directory)  
  
 modified: mars.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")

We can put things back the way they were by using git checkout:

$ git checkout HEAD mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity

As you might guess from its name, git checkout checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in HEAD, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

$ git checkout f22b25e mars.txt

$ cat mars.txt

Cold and dry, but everything is my favorite color

$ git status

# On branch master  
Changes to be committed:  
 (use "git reset HEAD <file>..." to unstage)  
# Changes not staged for commit:  
# (use "git add <file>..." to update what will be committed)  
# (use "git checkout -- <file>..." to discard changes in working directory)  
#  
# modified: mars.txt  
#  
no changes added to commit (use "git add" and/or "git commit -a")

Notice that the changes are on the staged area. Again, we can put things back the way they were with by using git checkout:

$ git checkout -f master mars.txt

**Don't Lose Your HEAD**

Above we used

$ git checkout f22b25e mars.txt

to revert mars.txt to its state after the commit f22b25e. If you forget mars.txt in that command, Git will tell you that "You are in 'detached HEAD' state." In this state, you shouldn't make any changes. You can fix this by reattaching your head using git checkout master

It's important to remember that we must use the commit number that identifies the state of the repository *before* the change we're trying to undo. A common mistake is to use the number of the commit in which we made the change we're trying to get rid of. In the example below, we want to retrieve the state from before the most recent commit (HEAD~1), which is commit f22b25e:

Git Checkout

Git Checkout

So, to put it all together, here's how Git works in cartoon form:

http://figshare.com/articles/How_Git_works_a_cartoon/1328266

<http://figshare.com/articles/How_Git_works_a_cartoon/1328266>

**Simplifying the Common Case**

If you read the output of git status carefully, you'll see that it includes this hint:

(use "git checkout -- <file>..." to discard changes in working directory)

As it says, git checkout without a version identifier restores files to the state saved in HEAD. The double dash -- is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

**Recovering Older Versions of a File**

Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning "broke" the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck...

Luckily, she has been keeping track of her project's versions using Git! Which commands below will let her recover the last committed version of her Python script called data\_cruncher.py?

1. $ git checkout HEAD
2. $ git checkout HEAD data\_cruncher.py
3. $ git checkout HEAD~1 data\_cruncher.py
4. $ git checkout <unique ID of last commit> data\_cruncher.py
5. Both 2 and 4

**Reverting a Commit**

Jennifer is collaborating on her Python script with her colleagues and realises her last commit to the group repository is wrong and wants to undo it. Jennifer needs to undo correctly so everyone in the group repository gets the correct change. git revert [wrong commit ID] will make a new commit that undoes Jennifer's previous wrong commit. Therefore git revert is different than git checkout [commit ID] because checkout is for local changes not committed to the group repository. Below are the right steps and explanations for Jennifer to use git revert, what is the missing command?

1. \_\_\_\_\_\_\_\_ # Look at the git history of the project to find the commit ID
2. Copy the ID (the first few characters of the ID, e.g. 0b1d055).
3. git revert [commit ID]
4. Type in the new commit message.
5. Save and close

**Understanding Workflow and History**

What is the output of cat venus.txt at the end of this set of commands?

$ cd planets  
$ nano venus.txt #input the following text: Venus is beautiful and full of love  
$ git add venus.txt  
$ nano venus.txt #add the following text: Venus is too hot to be suitable as a base  
$ git commit -m "comments on Venus as an unsuitable base"  
$ git checkout HEAD venus.txt  
$ cat venus.txt #this will print the contents of venus.txt to the screen

Venus is too hot to be suitable as a base

Venus is beautiful and full of love

Venus is beautiful and full of love  
Venus is too hot to be suitable as a base

Error because you have changed venus.txt without committing the changes

**Checking Understanding of git diff**

Consider this command: git diff HEAD~3 mars.txt. What do you predict this command will do if you execute it? What happens when you do execute it? Why?

Try another command, git diff [ID] mars.txt, where [ID] is replaced with the unique identifier for your most recent commit. What do you think will happen, and what does happen?

**Getting Rid of Staged Changes**

git checkout can be used to restore a previous commit when unstaged changes have been made, but will it also work for changes that have been staged but not committed? Make a change to mars.txt, add that change, and use git checkout to see if you can remove your change.

**Explore and Summarize Histories**

Exploring history is an important part of git, often it is a challenge to find the right commit ID, especially if the commit is from several months ago.

Imaging the planets project has more than 50 files. You would like to find a commit with specific text in mars.txt is modified. When you type git log, a very long list appeared, How can you narrow down the search?

Recorded that the git diff command allow us to explore one specific file, e.g. git diff mars.txt. We can apply the similar idea here.

$ git log mars.txt

Unfortunately some of these commit messages are very ambiguous e.g. update files. How can you search through these files?

Both git diff and git log are very useful and they summarize different part of the history for you. Is that possible to combine both? Let's try the following:

$ git log --patch mars.txt

You should get a long list of output, and you should be able to see both commit messages and the difference between each commit.

Question: What does the following command do?

$ git log --patch HEAD~3 HEAD~1 \*.txt

## Ignoring Things

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis. Let's create a few dummy files:

$ mkdir results  
$ touch a.dat b.dat c.dat results/a.out results/b.out

and see what Git says:

$ git status

On branch master  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
  
 a.dat  
 b.dat  
 c.dat  
 results/  
nothing added to commit but untracked files present (use "git add" to track)

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called .gitignore:

$ nano .gitignore  
$ cat .gitignore

\*.dat  
results/

These patterns tell Git to ignore any file whose name ends in .dat and everything in the results directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of git status is much cleaner:

$ git status

On branch master  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
  
 .gitignore  
nothing added to commit but untracked files present (use "git add" to track)

The only thing Git notices now is the newly-created .gitignore file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit .gitignore:

$ git add .gitignore  
$ git commit -m "Add the ignore file"  
$ git status

# On branch master  
nothing to commit, working directory clean

As a bonus, using .gitignore helps us avoid accidentally adding to the repository files that we don't want to track:

$ git add a.dat

The following paths are ignored by one of your .gitignore files:  
a.dat  
Use -f if you really want to add them.

If we really want to override our ignore settings, we can use git add -f to force Git to add something. For example, git add -f a.dat. We can also always see the status of ignored files if we want:

$ git status --ignored

On branch master  
Ignored files:  
 (use "git add -f <file>..." to include in what will be committed)  
  
 a.dat  
 b.dat  
 c.dat  
 results/  
  
nothing to commit, working directory clean

**Ignoring Nested Files**

Given a directory structure that looks like:

results/data  
results/plots

How would you ignore only results/plots and not results/data?

**Ignoring all data Files in a Directory**

Given a directory structure that looks like:

results/data/position/gps/a.data  
results/data/position/gps/b.data  
results/data/position/gps/c.data  
results/data/position/gps/info.txt  
results/plots

What's the shortest .gitignore rule you could write to ignore all .data files in result/data/position/gps? Do not ignore the info.txt.

**The Order of Rules**

Given a .gitignore file with the following contents:

\*.data  
!\*.data

What will be the result?

**Log Files**

You wrote a script that creates many intermediate log-files of the form log\_01, log\_02, log\_03, etc. You want to keep them but you do not want to track them through git.

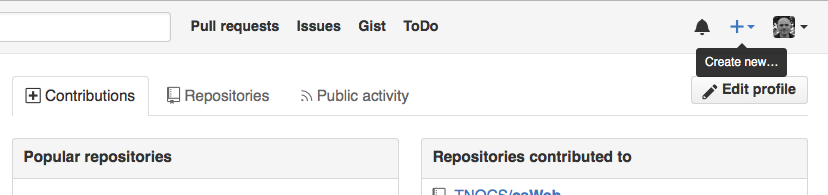
1. Write **one** .gitignore entry that excludes files of the form log\_01, log\_02, etc.
2. Test your "ignore pattern" by creating some dummy files of the form log\_01, etc.
3. You find that the file log\_01 is very important after all, add it to the tracked files without changing the .gitignore again.
4. Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via .gitignore.

## Remotes in GitHub

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

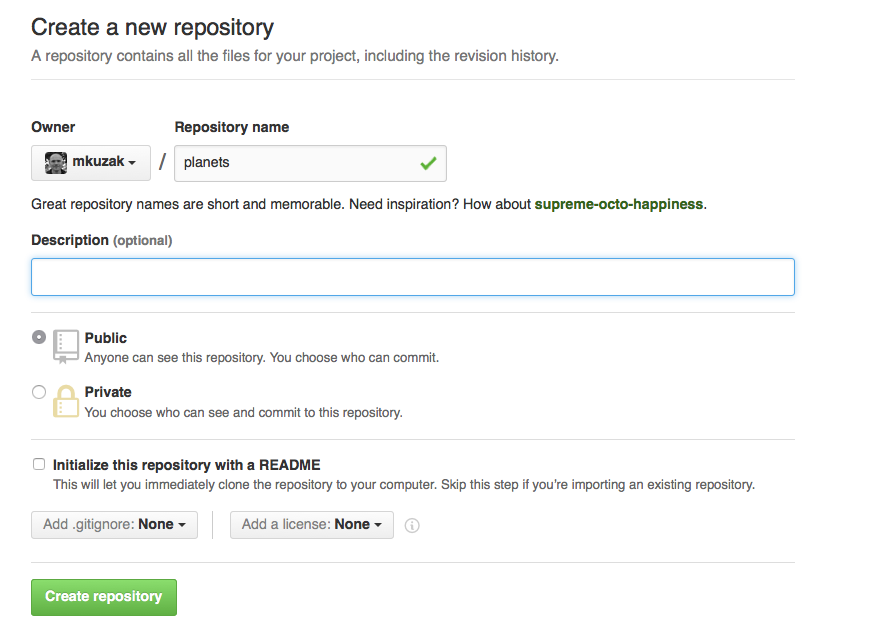
Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like [GitHub](http://github.com), [BitBucket](http://bitbucket.org) or [GitLab](http://gitlab.com/) to hold those master copies; we'll explore the pros and cons of this in the final section of this lesson.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository called planets:



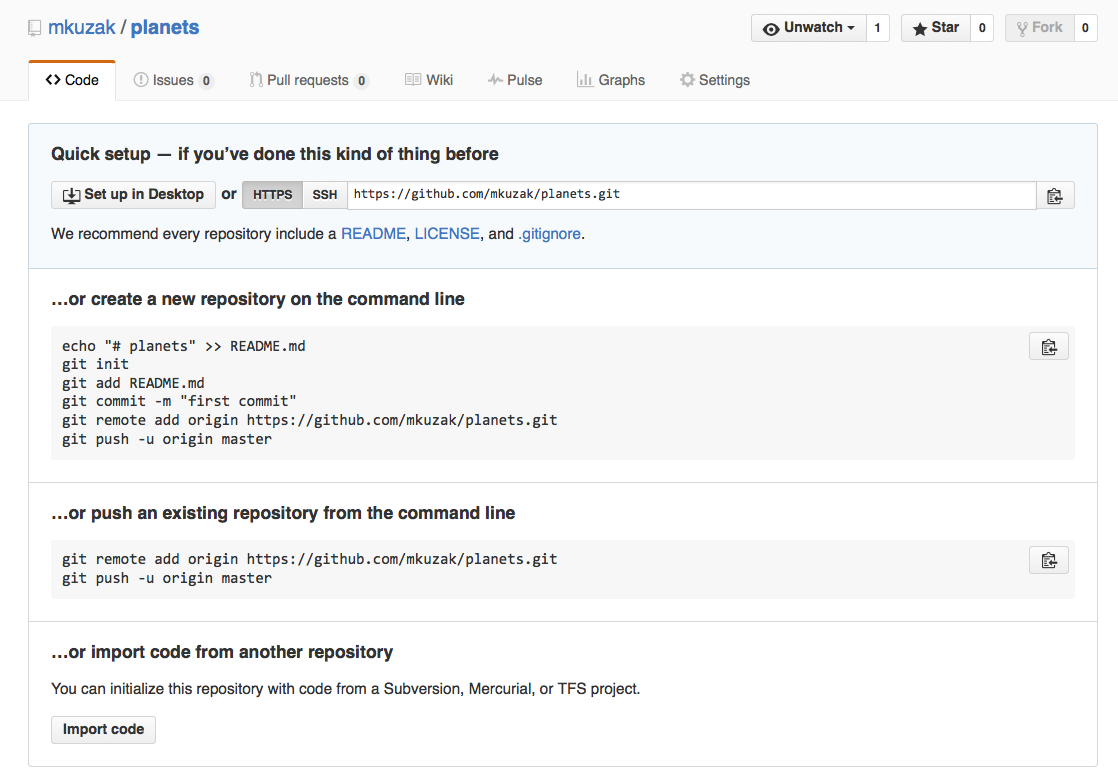
Creating a Repository on GitHub (Step 1)

Name your repository "planets" and then click "Create Repository":



Creating a Repository on GitHub (Step 2)

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository:



Creating a Repository on GitHub (Step 3)

This effectively does the following on GitHub's servers:

$ mkdir planets  
$ cd planets  
$ git init

Our local repository still contains our earlier work on mars.txt, but the remote repository on GitHub doesn't contain any files yet:

Freshly-Made GitHub Repository

Freshly-Made GitHub Repository

The next step is to connect the two repositories. We do this by making the GitHub repository a [remote](%7B%7B%20page.root%20%7D%7D/reference/#remote) for the local repository. The home page of the repository on GitHub includes the string we need to identify it:



Where to Find Repository URL on GitHub

Click on the 'HTTPS' link to change the [protocol](%7B%7B%20page.root%20%7D%7D/reference/#protocol) from SSH to HTTPS.

**HTTPS vs. SSH**

We use HTTPS here because it does not require additional configuration. After the workshop you may want to set up SSH access, which is a bit more secure, by following one of the great tutorials from [GitHub](https://help.github.com/articles/generating-ssh-keys), [Atlassian/BitBucket](https://confluence.atlassian.com/display/BITBUCKET/Set+up+SSH+for+Git) and [GitLab](https://about.gitlab.com/2014/03/04/add-ssh-key-screencast/) (this one has a screencast).



Changing the Repository URL on GitHub

Copy that URL from the browser, go into the local planets repository, and run this command:

$ git remote add origin https://github.com/vlad/planets.git

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of vlad.

We can check that the command has worked by running git remote -v:

$ git remote -v

origin https://github.com/vlad/planets.git (push)  
origin https://github.com/vlad/planets.git (fetch)

The name origin is a local nickname for your remote repository. We could use something else if we wanted to, but origin is by far the most common choice.

Once the nickname origin is set up, this command will push the changes from our local repository to the repository on GitHub:

$ git push origin master

Counting objects: 9, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (9/9), 821 bytes, done.  
Total 9 (delta 2), reused 0 (delta 0)  
To https://github.com/vlad/planets  
 \* [new branch] master -> master  
Branch master set up to track remote branch master from origin.

**Proxy**

If the network you are connected to uses a proxy there is an chance that your last command failed with "Could not resolve hostname" as the error message. To solve this issue you need to tell Git about the proxy:

$ git config --global http.proxy http://user:password@proxy.url  
$ git config --global https.proxy http://user:password@proxy.url

When you connect to another network that doesn't use a proxy you will need to tell Git to disable the proxy using:

$ git config --global --unset http.proxy  
$ git config --global --unset https.proxy

**Password Managers**

If your operating system has a password manager configured, git push will try to use it when it needs your username and password. For example, this is the default behavior for Git Bash on Windows. If you want to type your username and password at the terminal instead of using a password manager, type:

$ unset SSH\_ASKPASS

in the terminal, before you run git push. Despite the name, [git uses SSH\_ASKPASS for all credential entry](http://git-scm.com/docs/gitcredentials#_requesting_credentials), so you may want to unset SSH\_ASKPASS whether you are using git via SSH or https.

You may also want to add unset SSH\_ASKPASS at the end of your ~/.bashrc to make git default to using the terminal for usernames and passwords.

Our local and remote repositories are now in this state:

GitHub Repository After First Push

GitHub Repository After First Push

**The '-u' Flag**

You may see a -u option used with git push in some documentation. This option is synonymous with the --set-upstream-to option for the git branch command, and is used to associate the current branch with a remote branch so that the git pull command can be used without any arguments. To do this, simply use git push -u origin master once the remote has been set up.

We can pull changes from the remote repository to the local one as well:

$ git pull origin master

From https://github.com/vlad/planets  
 \* branch master -> FETCH\_HEAD  
Already up-to-date.

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

**GitHub GUI**

Browse to your planets repository on GitHub. Under the Code tab, find and click on the text that says "XX commits" (where "XX" is some number). Hover over, and click on, the three buttons to the right of each commit. What information can you gather/explore from these buttons? How would you get that same information in the shell?

**GitHub Timestamp**

Create a remote repository on GitHub. Push the contents of your local repository to the remote. Make changes to your local repository and push these changes. Go to the repo you just created on GitHub and check the [timestamps](%7B%7B%20page.root%20%7D%7D/reference/#timestamp) of the files. How does GitHub record times, and why?

**Push vs. Commit**

In this lesson, we introduced the "git push" command. How is "git push" different from "git commit"?

**Fixing Remote Settings**

It happens quite often in practice that you made a typo in the remote URL. This exercice is about how to fix this kind of issues. First start by adding a remote with an invalid URL:

git remote add broken https://github.com/this/url/is/invalid

Do you get an error when adding the remote? Can you think of a command that would make it obvious that your remote URL was not valid? Can you figure out how to fix the URL (tip: use git remote -h)? Don't forget to clean up and remove this remote once you are done with this exercise.

**GitHub License and README files**

In this section we learned about creating a remote repository on GitHub, but when you initialized your GitHub repo, you didn't add a README.md or a license file. If you had, what do you think would have happened when you tried to link your local and remote repositories?

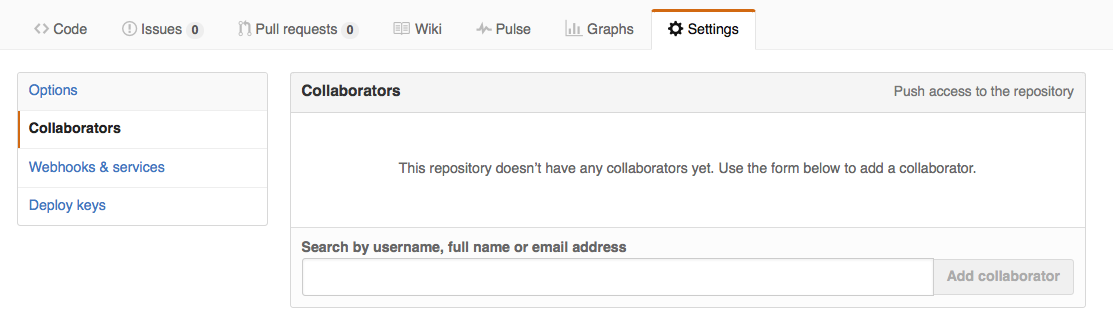
## Collaborating

For the next step, get into pairs. One person will be the "Owner" and the other will be the "Collaborator". The goal is that the Collaborator add changes into the Owner's repository. We will switch roles at the end, so both persons will play Owner and Collaborator.

**Practicing By Yourself**

If you're working through this lesson on your own, you can carry on by opening a second terminal window. This window will represent your partner, working on another computer. You won't need to give anyone access on GitHub, because both 'partners' are you.

The Owner needs to give the Collaborator access. On GitHub, click the settings button on the right, then select Collaborators, and enter your partner's username.



Adding Collaborators on GitHub

To accept access to the Owner's repo, the Collaborator needs to go to <https://github.com/notifications>. Once there she can accept access to the Owner's repo.

Next, the Collaborator needs to download a copy of the Owner's repository to her machine. This is called "cloning a repo". To clone the Owner's repo into her Desktop folder, the Collaborator enters:

$ git clone https://github.com/vlad/planets.git ~/Desktop/vlad-planets

Replace 'vlad' with the Owner's username.

After Creating Clone of Repository

After Creating Clone of Repository

The Collaborator can now make a change in her clone of the Owner's repository, exactly the same way as we've been doing before:

$ cd ~/Desktop/vlad-planets  
$ nano pluto.txt  
$ cat pluto.txt

It is so a planet!

$ git add pluto.txt  
$ git commit -m "Some notes about Pluto"

1 file changed, 1 insertion(+)  
 create mode 100644 pluto.txt

Then push the change to the *Owner's repository* on GitHub:

$ git push origin master

Counting objects: 4, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 306 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To https://github.com/vlad/planets.git  
 9272da5..29aba7c master -> master

Note that we didn't have to create a remote called origin: Git uses this name by default when we clone a repository. (This is why origin was a sensible choice earlier when we were setting up remotes by hand.)

Take a look to the Owner's repository on its GitHub website now (maybe you need to refresh your browser.) You should be able to see the new commit made by the Collaborator.

To download the Collaborator's changes from GitHub, the Owner now enters:

$ git pull origin master

remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 0), reused 3 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From https://github.com/vlad/planets  
 \* branch master -> FETCH\_HEAD  
Updating 9272da5..29aba7c  
Fast-forward  
 pluto.txt | 1 +  
 1 file changed, 1 insertion(+)  
 create mode 100644 pluto.txt

Now the three repositories (Owner's local, Collaborator's local, and Owner's on GitHub) are back in sync.

**A Basic Collaborative Workflow**

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should git pull before making our changes. The basic collaborative workflow would be:

* update your local repo with git pull origin master,
* make your changes and stage them with git add,
* commit your changes with git commit -m, and
* upload the changes to GitHub with git push origin master

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.

**Switch Roles and Repeat**

Switch roles and repeat the whole process.

**Review Changes**

The Owner push commits to the repository without giving any information to the Collaborator. How can the Collaborator find out what has changed with command line? And on GitHub?

**Comment Changes in GitHub**

The Collaborator has some questions about one line change made by the Owner and has some suggestions to propose.

With GitHub, it is possible to comment the diff of a commit. Over the line of code to comment, a blue comment icon appears to open a comment window.

The Collaborator posts its comments and suggestions using GitHub interface.

**Version History, Backup, and Version Control**

Some backup software can keep a history of the versions of your files. They also allows you to recover specific versions. How is this functionality different from version control? What are some of the benifits of using version control, Git and GitHub?

## Conflicts

As soon as people can work in parallel, it's likely someone's going to step on someone else's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these [conflicts](%7B%7B%20page.root%20%7D%7D/reference/#conflicts) by giving us tools to [resolve](%7B%7B%20page.root%20%7D%7D/reference/#resolve) overlapping changes.

To see how we can resolve conflicts, we must first create one. The file mars.txt currently looks like this in both partners' copies of our planets repository:

$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity

Let's add a line to one partner's copy only:

$ nano mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
This line added to Wolfman's copy

and then push the change to GitHub:

$ git add mars.txt  
$ git commit -m "Adding a line in our home copy"

[master 5ae9631] Adding a line in our home copy  
 1 file changed, 1 insertion(+)

$ git push origin master

Counting objects: 5, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 352 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To https://github.com/vlad/planets  
 29aba7c..dabb4c8 master -> master

Now let's have the other partner make a different change to their copy *without* updating from GitHub:

$ nano mars.txt  
$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
We added a different line in the other copy

We can commit the change locally:

$ git add mars.txt  
$ git commit -m "Adding a line in my copy"

[master 07ebc69] Adding a line in my copy  
 1 file changed, 1 insertion(+)

but Git won't let us push it to GitHub:

$ git push origin master

To https://github.com/vlad/planets.git  
 ! [rejected] master -> master (non-fast-forward)  
error: failed to push some refs to 'https://github.com/vlad/planets.git'  
hint: Updates were rejected because the tip of your current branch is behind  
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')  
hint: before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

The Conflicting Changes

The Conflicting Changes

Git detects that the changes made in one copy overlap with those made in the other and stops us from trampling on our previous work. What we have to do is pull the changes from GitHub, [merge](%7B%7B%20page.root%20%7D%7D/reference/#merge) them into the copy we're currently working in, and then push that. Let's start by pulling:

$ git pull origin master

remote: Counting objects: 5, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 1), reused 3 (delta 1)  
Unpacking objects: 100% (3/3), done.  
From https://github.com/vlad/planets  
 \* branch master -> FETCH\_HEAD  
Auto-merging mars.txt  
CONFLICT (content): Merge conflict in mars.txt  
Automatic merge failed; fix conflicts and then commit the result.

git pull tells us there's a conflict, and marks that conflict in the affected file:

$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
<<<<<<< HEAD  
We added a different line in the other copy  
=======  
This line added to Wolfman's copy  
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d

Our change—the one in HEAD—is preceded by <<<<<<<. Git has then inserted ======= as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with >>>>>>>. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
We removed the conflict on this line

To finish merging, we add mars.txt to the changes being made by the merge and then commit:

$ git add mars.txt  
$ git status

On branch master  
All conflicts fixed but you are still merging.  
 (use "git commit" to conclude merge)  
  
Changes to be committed:  
  
 modified: mars.txt

$ git commit -m "Merging changes from GitHub"

[master 2abf2b1] Merging changes from GitHub

Now we can push our changes to GitHub:

$ git push origin master

Counting objects: 10, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (6/6), 697 bytes, done.  
Total 6 (delta 2), reused 0 (delta 0)  
To https://github.com/vlad/planets.git  
 dabb4c8..2abf2b1 master -> master

Git keeps track of what we've merged with what, so we don't have to fix things by hand again when the collaborator who made the first change pulls again:

$ git pull origin master

remote: Counting objects: 10, done.  
remote: Compressing objects: 100% (4/4), done.  
remote: Total 6 (delta 2), reused 6 (delta 2)  
Unpacking objects: 100% (6/6), done.  
From https://github.com/vlad/planets  
 \* branch master -> FETCH\_HEAD  
Updating dabb4c8..2abf2b1  
Fast-forward  
 mars.txt | 2 +-  
 1 file changed, 1 insertion(+), 1 deletion(-)

We get the merged file:

$ cat mars.txt

Cold and dry, but everything is my favorite color  
The two moons may be a problem for Wolfman  
But the Mummy will appreciate the lack of humidity  
We removed the conflict on this line

We don't need to merge again because Git knows someone has already done that.

Git's ability to resolve conflicts is very useful, but conflict resolution costs time and effort, and can introduce errors if conflicts are not resolved correctly. If you find yourself resolving a lot of conflicts in a project, consider one of these approaches to reducing them:

* Try breaking large files apart into smaller files so that it is less likely that two authors will be working in the same file at the same time
* Clarify who is responsible for what areas with your collaborators
* Discuss what order tasks should be carried out in with your collaborators so that tasks that will change the same file won't be worked on at the same time

**Solving Conflicts that You Create**

Clone the repository created by your instructor. Add a new file to it, and modify an existing file (your instructor will tell you which one). When asked by your instructor, pull her changes from the repository to create a conflict, then resolve it.

**Conflicts on Non-textual files**

What does Git do when there is a conflict in an image or some other non-textual file that is stored in version control?

**A Typical Work Session**

You sit down at your computer to work on a shared project that is tracked in a remote Git repository. During your work session, you take the following actions, but not in this order:

* *Make changes* by appending the number 100 to a text file numbers.txt
* *Update remote* repository to match the local repository
* *Celebrate* your success with beer(s)
* *Update local* repository to match the remote repository
* *Stage changes* to be committed
* *Commit changes* to the local repository

In what order should you perform these actions to minimize the chances of conflicts? Put the commands above in order in the *action* column of the table below. When you have the order right, see if you can write the corresponding commands in the *command* column. A few steps are populated to get you started.

|  |  |  |
| --- | --- | --- |
| order | action . . . . . . . . . . | command . . . . . . . . . . |
| 1 |  |  |
| 2 |  | echo 100 >> numbers.txt |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 | Celebrate! | AFK |

## Open Science

The opposite of "open" isn't "closed". The opposite of "open" is "broken".

--- John Wilbanks

Free sharing of information might be the ideal in science, but the reality is often more complicated. Normal practice today looks something like this:

* A scientist collects some data and stores it on a machine that is occasionally backed up by her department.
* She then writes or modifies a few small programs (which also reside on her machine) to analyze that data.
* Once she has some results, she writes them up and submits her paper. She might include her data—a growing number of journals require this—but she probably doesn't include her code.
* Time passes.
* The journal sends her reviews written anonymously by a handful of other people in her field. She revises her paper to satisfy them, during which time she might also modify the scripts she wrote earlier, and resubmits.
* More time passes.
* The paper is eventually published. It might include a link to an online copy of her data, but the paper itself will be behind a paywall: only people who have personal or institutional access will be able to read it.

For a growing number of scientists, though, the process looks like this:

* The data that the scientist collects is stored in an open access repository like [figshare](http://figshare.com/) or [Zenodo](http://zenodo.org), possibly as soon as it's collected, and given its own [Digital Object Identifier](https://en.wikipedia.org/wiki/Digital_object_identifier) (DOI). Or the data was already published and is stored in [Dryad](http://datadryad.org/).
* The scientist creates a new repository on GitHub to hold her work.
* As she does her analysis, she pushes changes to her scripts (and possibly some output files) to that repository. She also uses the repository for her paper; that repository is then the hub for collaboration with her colleagues.
* When she's happy with the state of her paper, she posts a version to [arXiv](http://arxiv.org/) or some other preprint server to invite feedback from peers.
* Based on that feedback, she may post several revisions before finally submitting her paper to a journal.
* The published paper includes links to her preprint and to her code and data repositories, which makes it much easier for other scientists to use her work as starting point for their own research.

This open model accelerates discovery: the more open work is, [the more widely it is cited and re-used](http://dx.doi.org/10.1371/journal.pone.0000308). However, people who want to work this way need to make some decisions about what exactly "open" means and how to do it. You can find more on the different aspects of Open Science in [this book](http://link.springer.com/book/10.1007/978-3-319-00026-8).

This is one of the (many) reasons we teach version control. When used diligently, it answers the "how" question by acting as a shareable electronic lab notebook for computational work:

* The conceptual stages of your work are documented, including who did what and when. Every step is stamped with an identifier (the commit ID) that is for most intents and purposes unique.
* You can tie documentation of rationale, ideas, and other intellectual work directly to the changes that spring from them.
* You can refer to what you used in your research to obtain your computational results in a way that is unique and recoverable.
* With a distributed version control system such as Git, the version control repository is easy to archive for perpetuity, and contains the entire history.

**Making Code Citable**

[This short guide](https://guides.github.com/activities/citable-code/) from GitHub explains how to create a Digital Object Identifier (DOI) for your code, your papers, or anything else hosted in a version control repository.

**How Reproducible Is My Work?**

Ask one of your labmates to reproduce a result you recently obtained using only what they can find in your papers or on the web. Try to do the same for one of their results, then try to do it for a result from a lab you work with.

**How to Find an Appropriate Data Repository?**

Surf the internet for a couple of minutes and check out the data repositories mentioned above: [Figshare](http://figshare.com/), [Zenodo](http://zenodo.org), [Dryad](http://datadryad.org/). Depending on your field of research, you might find community-recognized repositories that are well-known in your field. You might also find useful [these data repositories recommended by Nature](http://www.nature.com/sdata/data-policies/repositories). Discuss with your neighbor which data repository you might want to approach for your current project and explain why.

**Can I Also Publish Code?**

There are many new ways to publish code and to make it citable. One way is described [on the homepage of GitHub itself](https://guides.github.com/activities/citable-code/). Basically it's a combination of GitHub (where the code is) and Zenodo (the repository creating the DOI). Read through this page while being aware that this is only one of many ways to making your code citable.

## Licensing

When a repository with source code, a manuscript or other creative works becomes public, it should include a file LICENSE or LICENSE.txt in the base directory of the repository that clearly states under which license the content is being made available. This is because creative works are automatically eligible for intellectual property (and thus copyright) protection. Reusing creative works without a license is dangerous, because the copyright holders could sue you for copyright infringement.

A license solves this problem by granting rights to others (the licensees) that they would otherwise not have. What rights are being granted under which conditions differs, often only slightly, from one license to another. In practice, a few licenses are by far the most popular, and [choosealicense.com](http://choosealicense.com/) will help you find a common license that suits your needs. Important considerations include:

* Whether you want to address patent rights.
* Whether you require people distributing derivative works to also distribute their source code.
* Whether the content you are licensing is source code.
* Whether you want to license the code at all.

Choosing a licence that is in common use makes life easier for contributors and users, because they are more likely to already be familiar with the license and don't have to wade through a bunch of jargon to decide if they're ok with it. The [Open Source Inititative](http://opensource.org/licenses) and [Free Software Foundation](http://www.gnu.org/licenses/license-list.html) both maintain lists of licenses which are good choices.

[This article](http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002598) provides an excellent overview of licensing and licensing options from the perspective of scientists who also write code.

At the end of the day what matters is that there is a clear statement as to what the license is. Also, the license is best chosen from the get-go, even if for a repository that is not public. Pushing off the decision only makes it more complicated later, because each time a new collaborator starts contributing, they, too, hold copyright and will thus need to be asked for approval once a license is chosen.

**Can I Use Open License?**

Find out whether you are allowed to apply an open license to your software. Can you do this unilaterally, or do you need permission from someone in your institution? If so, who?

**What licenses have I already accepted?**

Many of the software tools we use on a daily basis (including in this workshop) are released as open-source software. Pick a project on GitHub from the list below, or one of your own choosing. Find its license (usually in a file called LICENSE or COPYING) and talk about how it restricts your use of the software. Is it one of the licenses discussed in this session? How is it different? - [Git](https://github.com/git/git), the source-code management tool - [CPython](https://github.com/python/cpython), the standard implementation of the Python language - [Jupyter](https://github.com/jupyter), the project behind the web-based Python notebooks we'll be using - [EtherPad](https://github.com/ether/etherpad-lite), a real-time collaborative editor

## Citation

You may want to include a file called CITATION or CITATION.txt that describes how to reference your project; the [one for Software Carpentry](https://github.com/swcarpentry/website/blob/gh-pages/CITATION) states:

To reference Software Carpentry in publications, please cite both of the following:  
  
Greg Wilson: "Software Carpentry: Getting Scientists to Write Better  
Code by Making Them More Productive". Computing in Science &  
Engineering, Nov-Dec 2006.  
  
Greg Wilson: "Software Carpentry: Lessons Learned". arXiv:1307.5448,  
July 2013.  
  
@article{wilson-software-carpentry-2006,  
 author = {Greg Wilson},  
 title = {Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive},  
 journal = {Computing in Science \& Engineering},  
 month = {November--December},  
 year = {2006},  
}  
  
@online{wilson-software-carpentry-2013,  
 author = {Greg Wilson},  
 title = {Software Carpentry: Lessons Learned},  
 version = {1},  
 date = {2013-07-20},  
 eprinttype = {arxiv},  
 eprint = {1307.5448}  
}

## Hosting

The second big question for groups that want to open up their work is where to host their code and data. One option is for the lab, the department, or the university to provide a server, manage accounts and backups, and so on. The main benefit of this is that it clarifies who owns what, which is particularly important if any of the material is sensitive (i.e., relates to experiments involving human subjects or may be used in a patent application). The main drawbacks are the cost of providing the service and its longevity: a scientist who has spent ten years collecting data would like to be sure that data will still be available ten years from now, but that's well beyond the lifespan of most of the grants that fund academic infrastructure.

Another option is to purchase a domain and pay an Internet service provider (ISP) to host it. This gives the individual or group more control, and sidesteps problems that can arise when moving from one institution to another, but requires more time and effort to set up than either the option above or the option below.

The third option is to use a public hosting service like [GitHub](http://github.com), [GitLab](http://gitlab.com), [BitBucket](http://bitbucket.org), or [SourceForge](http://sourceforge.net). Each of these services provides a web interface that enables people to create, view, and edit their code repositories. These services also provide communication and project management tools including issue tracking, wiki pages, email notifications, and code reviews. These services benefit from economies of scale and network effects: it's easier to run one large service well than to run many smaller services to the same standard. It's also easier for people to collaborate. Using a popular service can help connect your project with communities already using the same service.

As an example, Software Carpentry [is on GitHub](%7B%7B%20swc_github%20%7D%7D) where you can find the [source for this page](%7B%7Bpage.root%7D%7D/_episodes/13-hosting.md). Anyone with a GitHub account can suggest changes to this text.

Using large, well-established services can also help you quickly take advantage of powerful tools. One such tool, continuous integration (CI), can automatically run software builds and tests whenever code is committed or pull requests are submitted. Direct integration of CI with an online hosting service means this information is present in any pull request, and helps maintain code integrity and quality standards. While CI is still available in self-hosted situations, there is much less setup and maintenance involved with using an online service. Furthermore, such tools are often provided free of charge to open source projects, and are also available for private repositories for a fee.

**Institutional Barriers**

Sharing is the ideal for science, but many institutions place restrictions on sharing, for example to protect potentially patentable intellectual property. If you encounter such restrictions, it can be productive to inquire about the underlying motivations either to request an exception for a specific project or domain, or to push more broadly for institutional reform to support more open science.

**Can My Work Be Public?**

Find out whether you are allowed to host your work openly on a public forge. Can you do this unilaterally, or do you need permission from someone in your institution? If so, who?

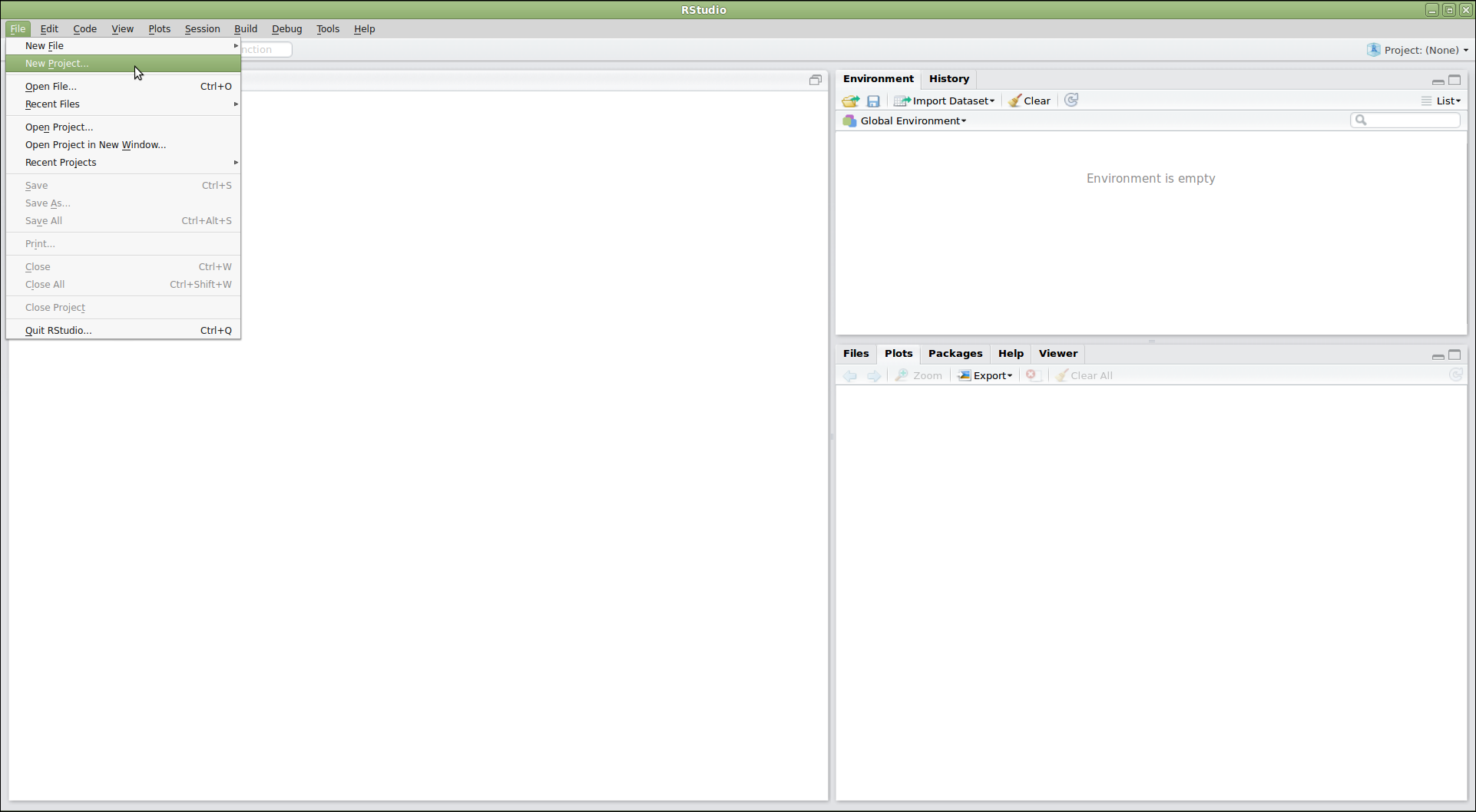
**Where Can I Share My Work?**

Does your institution have a repository or repositories that you can use to share your papers, data and software? How do institutional repositories differ from services like [arXiV](http://arxiv.org/), [figshare](http://figshare.com/) and [GitHub](http://github.com/)?

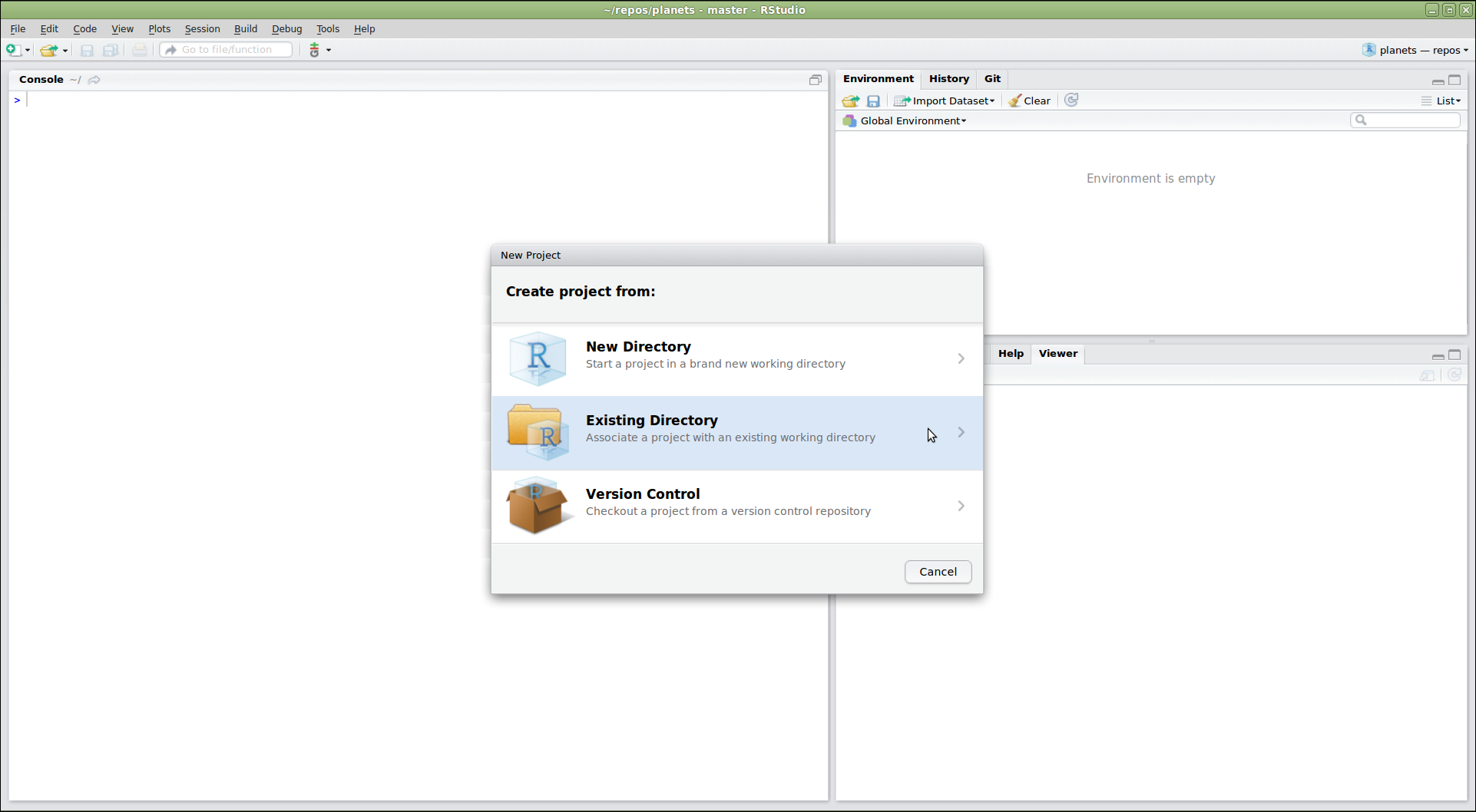
## Using Git from RStudio

Since version control is so useful when developing scripts, RStudio has built-in integration with Git. There are some more obscure Git features that you still need to use the command-line for, but RStudio has a nice interface for most common operations.

RStudio let's you create a [project](https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects) associated with a given directory. This is a way to keep track of related files. One of the way to keep track of them is via version control! To get started using RStudio for version control, let's make a new project:



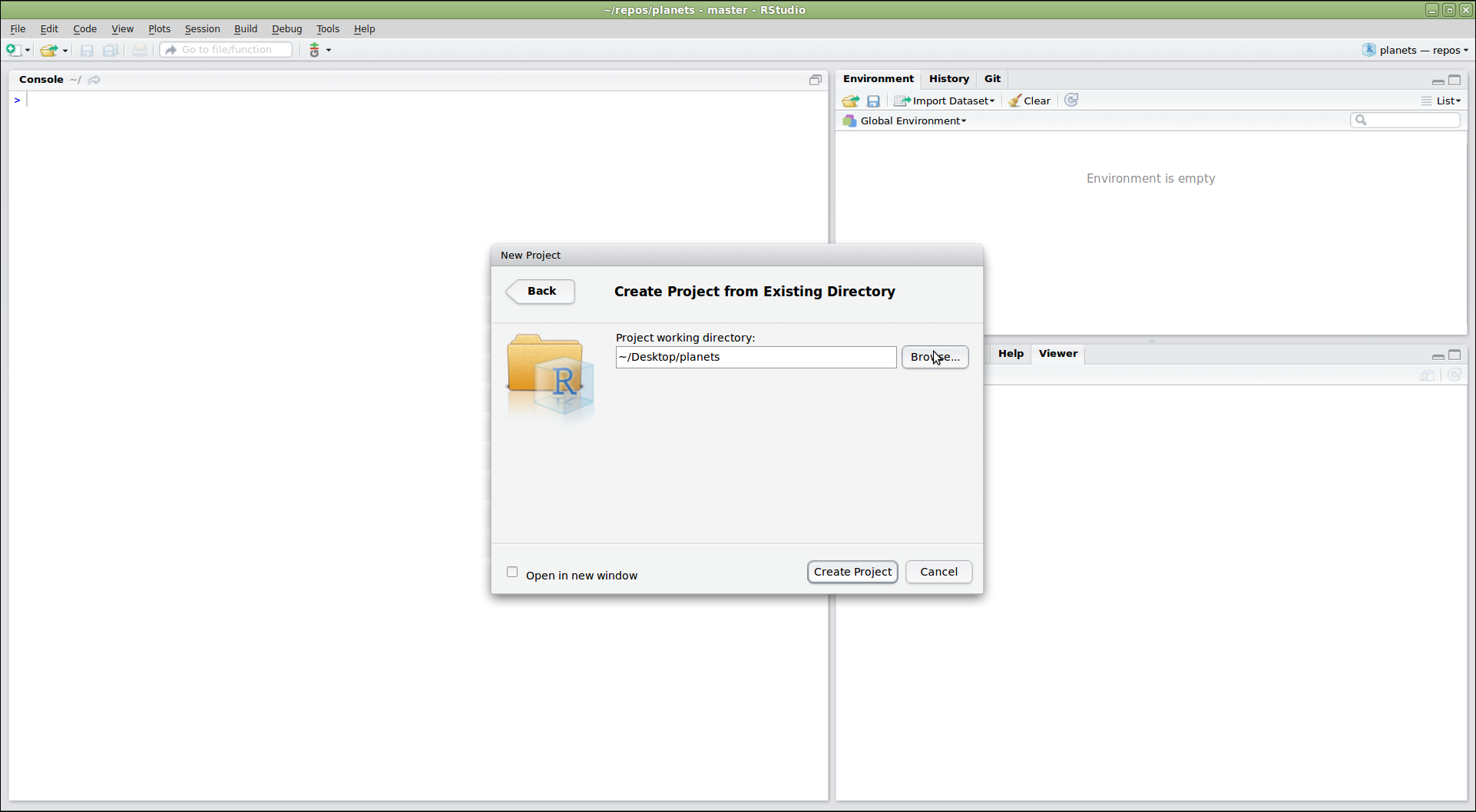
This will pop up a window asking us how we want to create the project. We have some options here. Let's say that we want to use RStudio with the planets repository that we already made. Since that repository lives in a directory on our computer, we'll choose "existing directory":



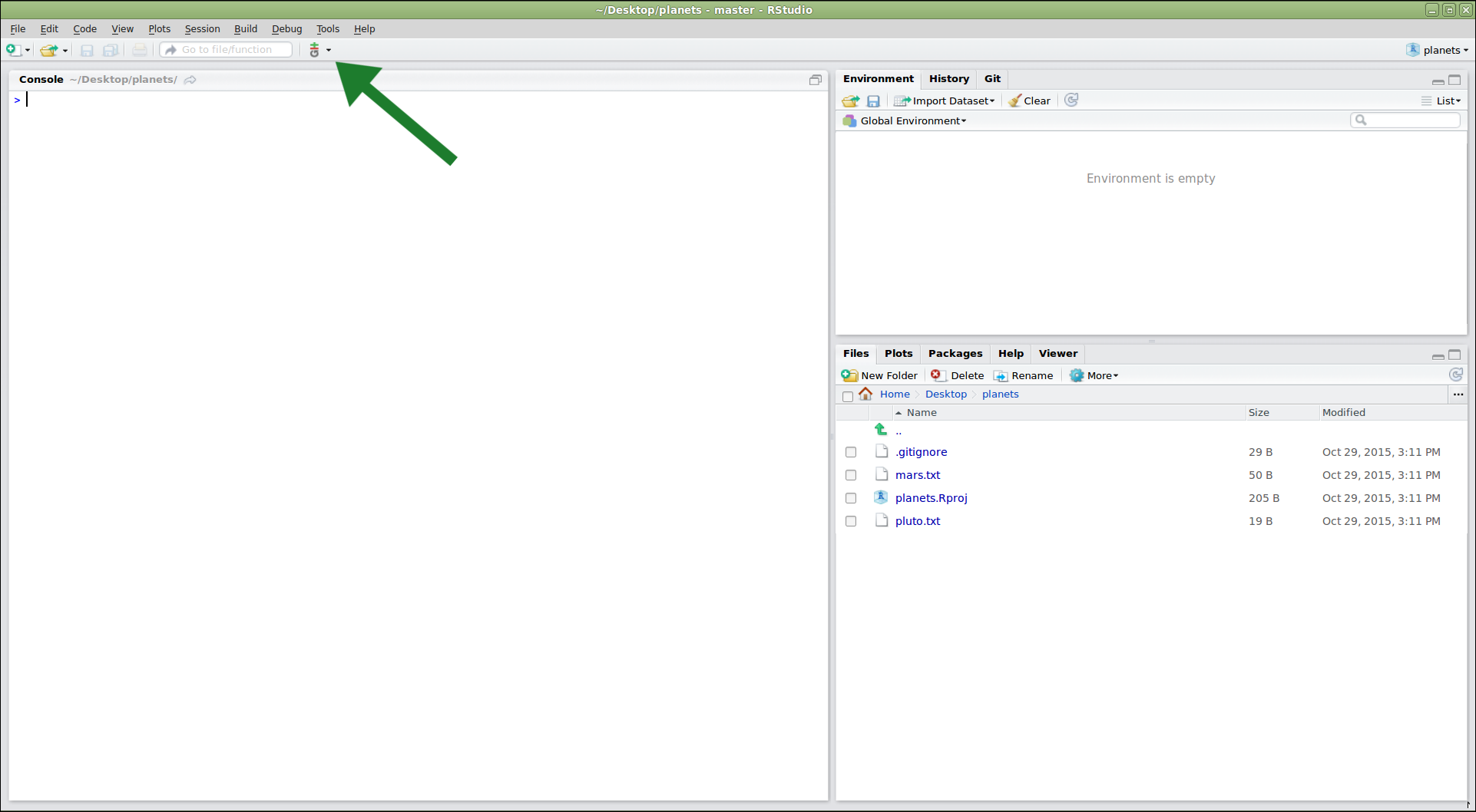
**Do You See a "Version Control" Option?**

Although we're not going to use it here, there should be a "version control" option on this menu. That is what you would click on if you wanted to create a project on your computer by cloning a repository from github. If that option is not present, it probably means that RStudio doesn't know where your Git executable is. See [this page](https://stat545-ubc.github.io/git03_rstudio-meet-git.html) for some debugging advice. Even if you have Git installed, you may need to accept the XCode license if you are using MacOSX.

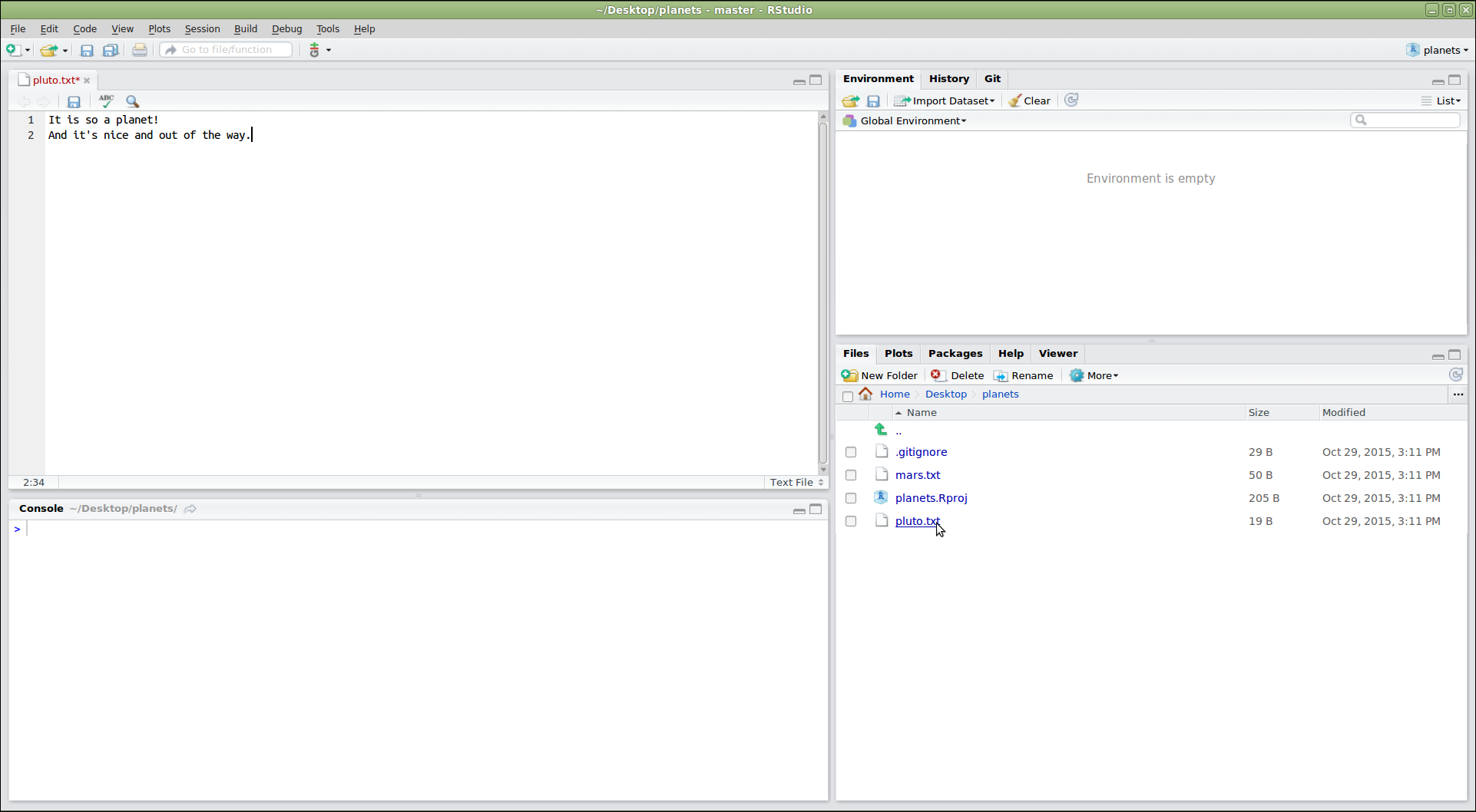
Next, RStudio will ask which existing directory we want to use. Click "browse" to navigate to the correct directory on your computer, then click "create project":



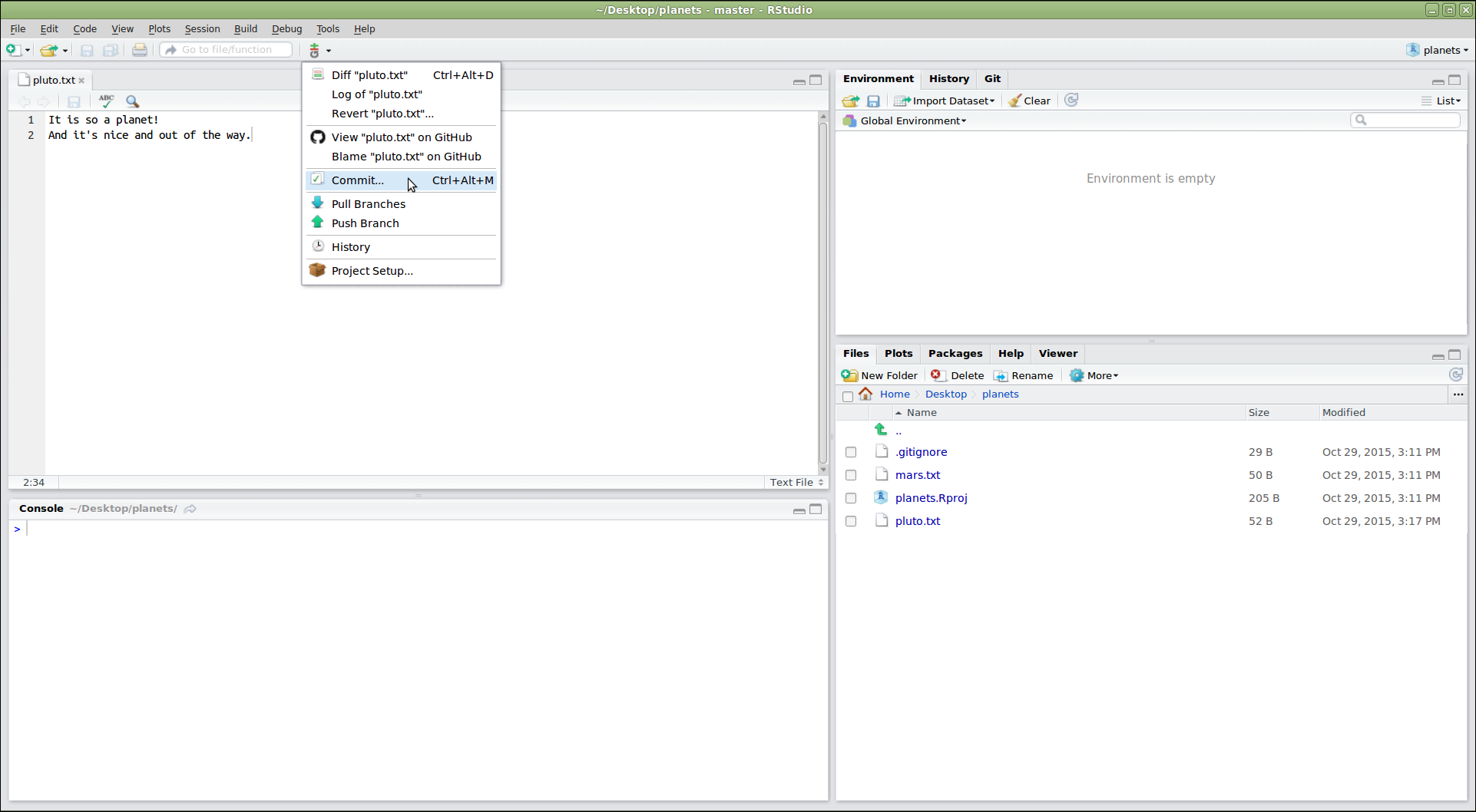
Ta-da! Now you have an R project containing your repository. Notice the vertical "Git" menu that is now on the menu bar. This means RStudio has recognized that this directory is a git repository, so it's giving you tools to use Git:



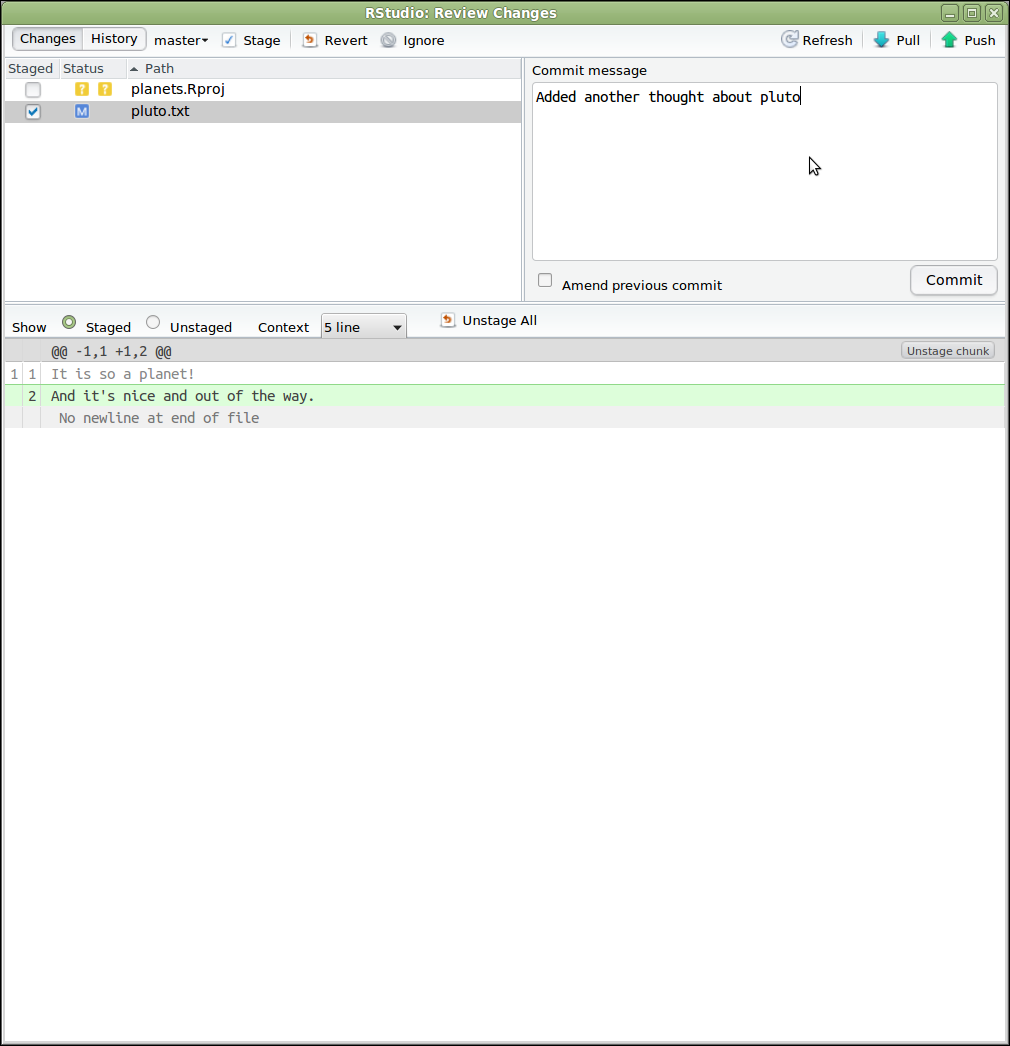
To edit the files in your repository, you can click on them from the panel in the lower right. Let's add some more information about pluto:



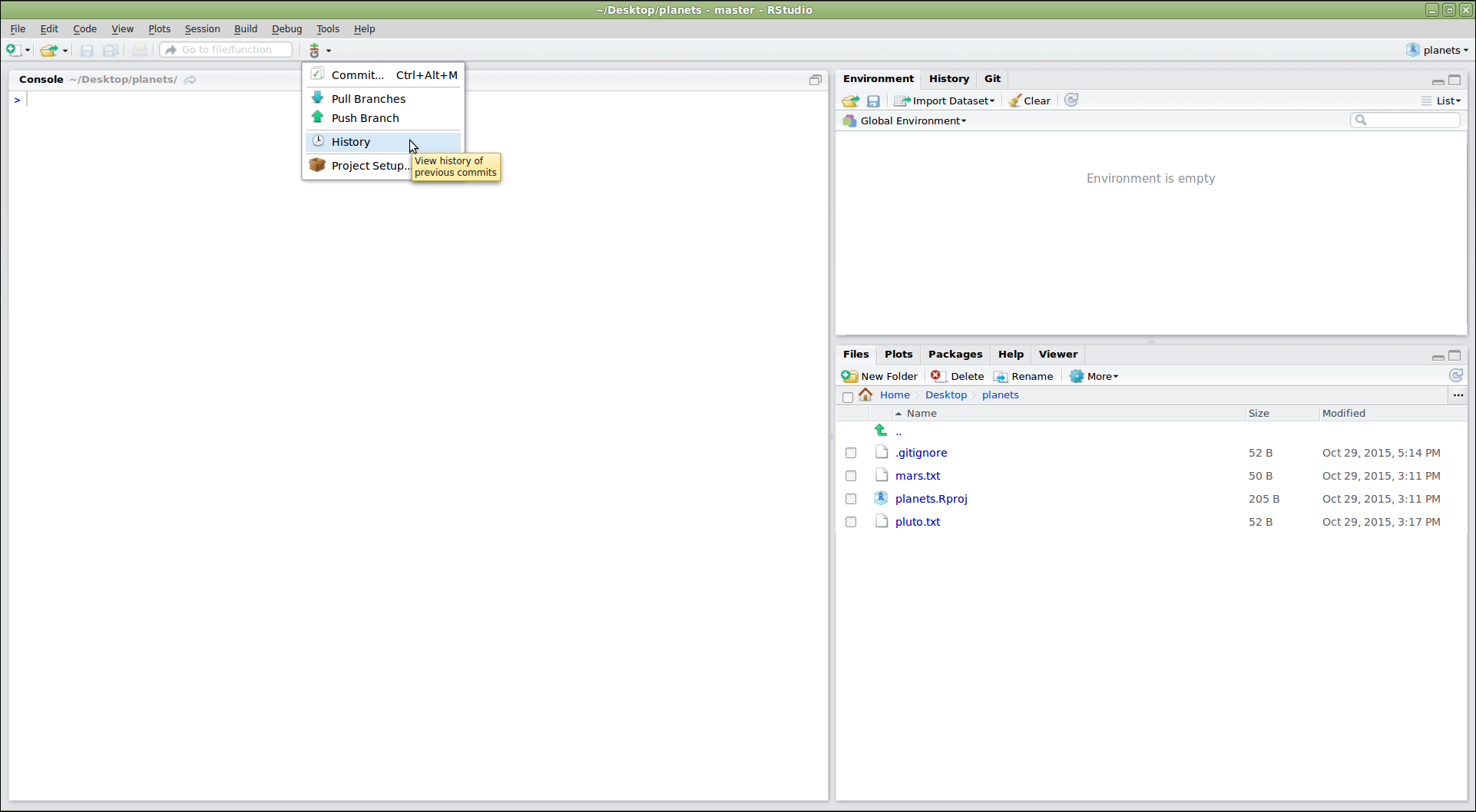
We can also use RStudio to commit these changes. Go to the git menu and click "commit":



This will bring up a screen where you can select which files to commit (check the boxes in the "staged" column) and enter a commit message (in the upper right). The icons in the "status" column indicate the current status of each file. You can also see the changes to each file by clicking on its name. Once everything is the way you want it, click "commit":



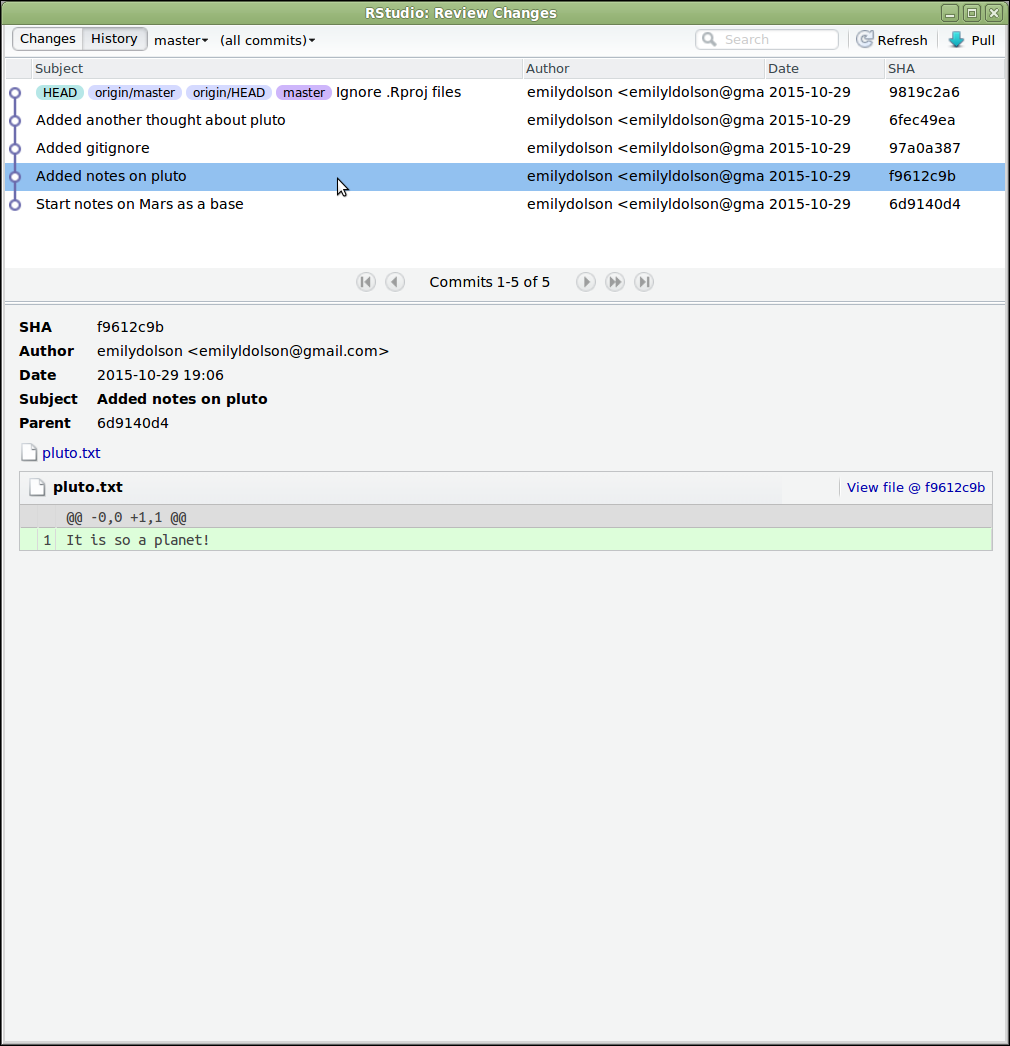
You can push these changes by selecting "push" from the Git menu. There are also options there to pull from a remote version of the repository, and view the history:



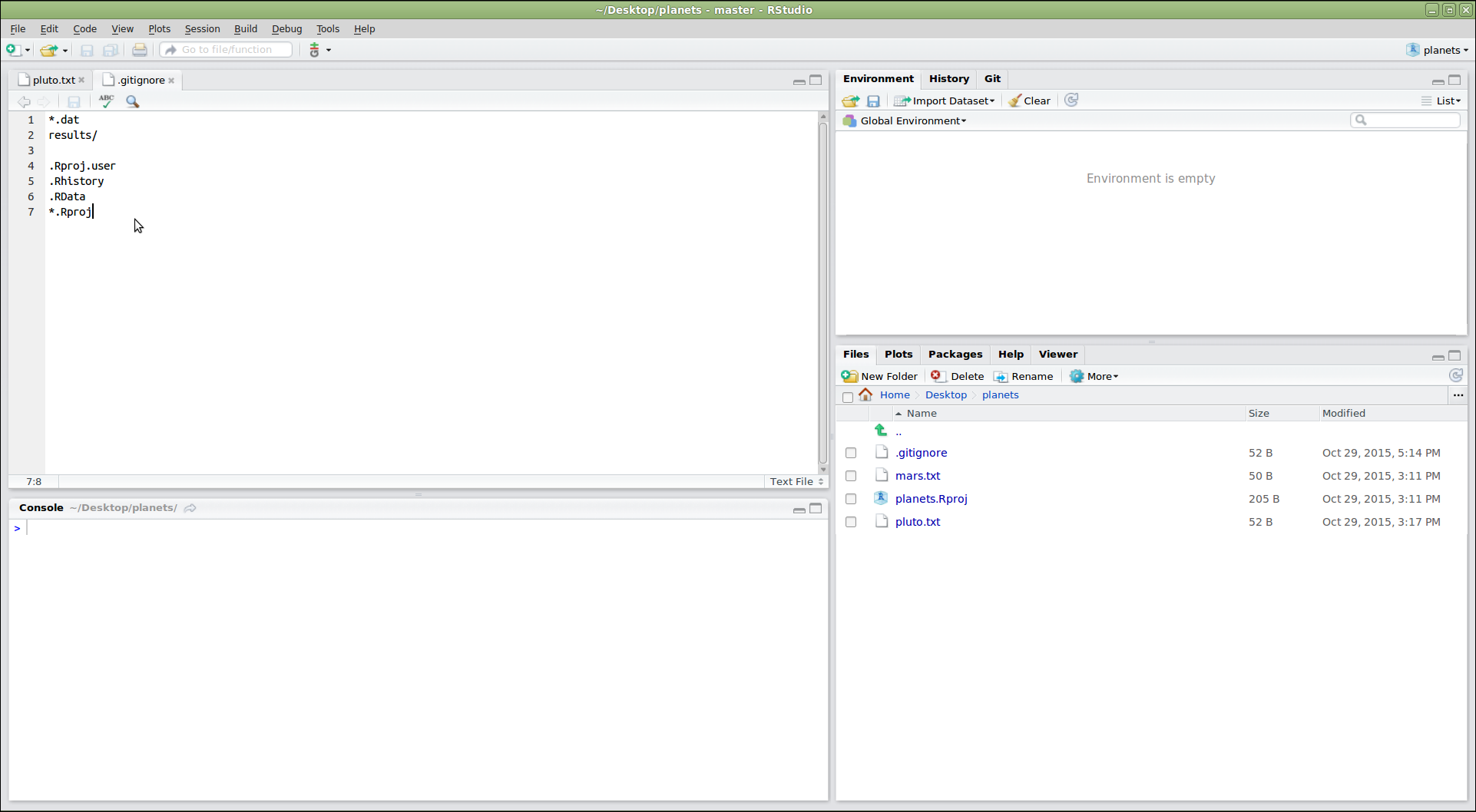
**Are the Push/Pull Commands Grayed Out?**

If this is the case, it generally means that RStudio doesn't know the location of any other version of your repository (i.e. the one on GitHub). To fix this, open a terminal to the repository and enter the command: git push -u origin master. Then restart RStudio.

If we click on "history", we can see a pretty graphical version of what git log would tell us:



RStudio creates some files that is uses to keep track of your project. You generally don't want to track these, so adding them to your .gitignore file is a good idea:



There are many more features buried in the RStudio git interface, but these should be enough to get you started!

## Learn more

1. "Happy Git with R": a user friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online: <http://happygitwithr.com>

# Import

## Readings

1. [W] Chapter 10, 11

## Tibbles vs. data.frame

### 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](https://github.com/hadley/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 homogenous 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.

In [Programming with R](#programming-with-r) we use data.frame in base R as our main data structure. From now on we will start to use tibble and other functions in [tidyverse](https://github.com/hadley/tidyverse) as much as possible. This will provide a special type of data frame called a "tibble" that has nice default printing behavior, among other benefits such as speed performance and better default behavior.

First, install and load tidyverse packages if you haven't yet:

install.packages("tidyverse")  
library(tidyverse)

There are two main differences in the usage of a tibble vs. a classic data.frame: printing and subsetting.

### Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from str():

tibble(  
 a = lubridate::now() + runif(1e3) \* 86400,  
 b = lubridate::today() + runif(1e3) \* 30,  
 c = 1:1e3,  
 d = runif(1e3),  
 e = sample(letters, 1e3, replace = TRUE)  
)

## # A tibble: 1,000 × 5  
## a b c d e  
## <dttm> <date> <int> <dbl> <chr>  
## 1 2017-03-16 04:30:06 2017-03-20 1 0.6678915 u  
## 2 2017-03-15 23:01:43 2017-03-18 2 0.6929406 n  
## 3 2017-03-15 13:32:41 2017-03-19 3 0.6695199 a  
## 4 2017-03-16 08:36:26 2017-03-25 4 0.2554328 a  
## 5 2017-03-15 15:48:02 2017-03-18 5 0.7300070 s  
## 6 2017-03-16 01:31:14 2017-03-22 6 0.9227604 d  
## 7 2017-03-15 19:33:27 2017-03-29 7 0.5513169 z  
## 8 2017-03-16 09:44:42 2017-04-06 8 0.5057338 p  
## 9 2017-03-15 20:23:31 2017-04-13 9 0.6074893 j  
## 10 2017-03-15 22:24:59 2017-04-07 10 0.6509957 y  
## # ... with 990 more rows

lubridate is a R package for date and time. You need to install the package if you want to replicate the above code on your own computer.

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly print() the data frame and control the number of rows (n) and the width of the display. width = Inf will display all columns:

print(mtcars, n = 10, width = Inf)

You can also control the default print behaviour by setting options:

* options(tibble.print\_max = n, tibble.print\_min = m): if more than m rows, print only n rows. Use options(dplyr.print\_min = Inf) to always show all rows.
* Use options(tibble.width = Inf) to always print all columns, regardless of the width of the screen.

You can see a complete list of options by looking at the package help with package?tibble.

A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations.

View(mtcars)

### Subsetting

So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, $ and [[. [[ can extract by name or column position; $ only extracts by name but is a little less typing.

df <- tibble(  
 x = runif(5),  
 y = rnorm(5)  
)  
  
# Extract by name  
df$x

## [1] 0.2181571 0.2636632 0.1628597 0.9998396 0.6868778

df[["x"]]

## [1] 0.2181571 0.2636632 0.1628597 0.9998396 0.6868778

# Extract by column position  
df[[1]]

## [1] 0.2181571 0.2636632 0.1628597 0.9998396 0.6868778

Compared to a data.frame, tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

### Converting

Some older functions don't work with tibbles. If you encounter one of these functions, use as.data.frame() to turn a tibble back to a data.frame:

class(as.data.frame(df))

## [1] "data.frame"

Or use as\_tibble() to convert a data.frame to tibble:

class(as\_tibble(mtcars))

## [1] "tbl\_df" "tbl" "data.frame"

The main reason that some older functions don't work with tibble is the [ function. We don't use [ much in this book because dplyr::filter() and dplyr::select() allow you to solve the same problems with clearer code (but you will learn a little about it in [vector subsetting](#vector-subsetting)). With base R data frames, [ sometimes returns a data frame, and sometimes returns a vector. With tibbles, [ always returns another tibble.

## Data Import

The most common data in data science is rectangular, spreadsheet-y data that works best to be loaded as a R data frame, and this section focus on import data into R as data frames. tidyverse provides the [readr](https://github.com/tidyverse/readr) package that reads comma-separated values (csv), tab delimited values (tsv), and fixed width files (fwf) as tibbles. Other R packages provide access to Excel spreadsheets, binary statistical data formats (SPSS, SAS, Stata), and relational databases:

* [readr](https://github.com/tidyverse/readr), reads flat files (csv, tsv, fwf) into R
* [readxl](https://github.com/tidyverse/readxl), reads excel files (.xls and .xlsx) into R
* [heaven](https://github.com/tidyverse/haven), reads SPSS, Stata and SAS files into R
* [rstats-db](https://github.com/rstats-db), R interface to databases
* [RMySQL](https://github.com/rstats-db/RMySQL)
* [RSQLite](https://github.com/rstats-db/RSQLite)
* [RPostgres](https://github.com/rstats-db/RPostgres)
* ...

For this section, we primarily focus on importing data in flat files into data frames. readr provides these functions:

* read\_csv() reads comma delimited files, read\_csv2() reads semicolon separated files (common in countries where , is used as the decimal place), read\_tsv() reads tab delimited files, and read\_delim() reads in files with any delimiter.
* read\_fwf() reads fixed width files. You can specify fields either by their widths with fwf\_widths() or their position with fwf\_positions(). read\_table() reads a common variation of fixed width files where columns are separated by white space.
* read\_log() reads Apache style log files. (But also check out [webreadr](https://github.com/Ironholds/webreadr) which is built on top of read\_log() and provides many more helpful tools.)

These functions all have similar syntax: once you've mastered one, you can use the others with ease. For the rest of this chapter we'll focus on read\_csv(). Not only are csv files one of the most common forms of data storage, but once you understand read\_csv(), you can easily apply your knowledge to all the other functions in readr.

The first argument to read\_csv() is the most important: it's the path to the file to read.

heights <- read\_csv("data/heights.csv")

## Parsed with column specification:  
## cols(  
## earn = col\_double(),  
## height = col\_double(),  
## sex = col\_character(),  
## ed = col\_integer(),  
## age = col\_integer(),  
## race = col\_character()  
## )

heights

## # A tibble: 1,192 × 6  
## earn height sex ed age race  
## <dbl> <dbl> <chr> <int> <int> <chr>  
## 1 50000 74.42444 male 16 45 white  
## 2 60000 65.53754 female 16 58 white  
## 3 30000 63.62920 female 16 29 white  
## 4 50000 63.10856 female 16 91 other  
## 5 51000 63.40248 female 17 39 white  
## 6 9000 64.39951 female 15 26 white  
## 7 29000 61.65633 female 12 49 white  
## 8 32000 72.69854 male 17 46 white  
## 9 2000 72.03947 male 15 21 hispanic  
## 10 27000 72.23493 male 12 26 white  
## # ... with 1,182 more rows

When you run read\_csv() it prints out a column specification that gives the name and type of each column. That's an important part of readr, which we'll come back to in [parsing a file].

You can also supply an inline csv file. This is useful for experimenting with readr and for creating reproducible examples to share with others:

read\_csv("a,b,c  
1,2,3  
4,5,6")

## # A tibble: 2 × 3  
## a b c  
## <int> <int> <int>  
## 1 1 2 3  
## 2 4 5 6

In both cases read\_csv() uses the first line of the data for the column names, which is a very common convention. There are two cases where you might want to tweak this behaviour:

1. Sometimes there are a few lines of metadata at the top of the file. You can use skip = n to skip the first n lines; or use comment = "#" to drop all lines that start with (e.g.) #.

* read\_csv("The first line of metadata  
   The second line of metadata  
   x,y,z  
   1,2,3", skip = 2)
* ## # A tibble: 1 × 3  
  ## x y z  
  ## <int> <int> <int>  
  ## 1 1 2 3
* read\_csv("# A comment I want to skip  
   x,y,z  
   1,2,3", comment = "#")
* ## # A tibble: 1 × 3  
  ## x y z  
  ## <int> <int> <int>  
  ## 1 1 2 3

1. The data might not have column names. You can use col\_names = FALSE to tell read\_csv() not to treat the first row as headings, and instead label them sequentially from X1 to Xn:

* read\_csv("1,2,3\n4,5,6", col\_names = FALSE)
* ## # A tibble: 2 × 3  
  ## X1 X2 X3  
  ## <int> <int> <int>  
  ## 1 1 2 3  
  ## 2 4 5 6
* ("\n" is a convenient shortcut for adding a new line. You'll learn more about it and other types of string escape in [string basics].)
* Alternatively you can pass col\_names a character vector which will be used as the column names:
* read\_csv("1,2,3\n4,5,6", col\_names = c("x", "y", "z"))
* ## # A tibble: 2 × 3  
  ## x y z  
  ## <int> <int> <int>  
  ## 1 1 2 3  
  ## 2 4 5 6

Another option that commonly needs tweaking is na: this specifies the value (or values) that are used to represent missing values in your file:

read\_csv("a,b,c\n1,2,.", na = ".")

## # A tibble: 1 × 3  
## a b c  
## <int> <int> <chr>  
## 1 1 2 <NA>

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab separated files with read\_tsv() and fixed width files with read\_fwf(). To read in more challenging files, you'll need to learn more about how readr parses each column, turning them into R vectors.

### Compared to base R

If you've used R before, you might wonder why we're not using read.csv(). There are a few good reasons to favour readr functions over the base equivalents:

* They are typically much faster (~10x) than their base equivalents. Long running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try data.table::fread(). It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.
* They produce tibbles, they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
* They are more reproducible. Base R functions inherit some behaviour from your operating system and environment variables, so import code that works on your computer might not work on someone else's.
* If you're interested in learning more on under the hood magics of how readr parses file, [this section](http://r4ds.had.co.nz/data-import.html#parsing-a-vector) in R for Data Science provides an overview.

### Exercises

1. What function would you use to read a file where fields were separated with  
   "|"?
2. Apart from file, skip, and comment, what other arguments do read\_csv() and read\_tsv() have in common?
3. What are the most important arguments to read\_fwf()?
4. Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like " or '. By convention, read\_csv() assumes that the quoting character will be ", and if you want to change it you'll need to use read\_delim() instead. What arguments do you need to specify to read the following text into a data frame?

* "x,y\n1,'a,b'"

1. Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

* read\_csv("a,b\n1,2,3\n4,5,6")  
  read\_csv("a,b,c\n1,2\n1,2,3,4")  
  read\_csv("a,b\n\"1")  
  read\_csv("a,b\n1,2\na,b")  
  read\_csv("a;b\n1;3")

## Writing to a file

readr also comes with two useful functions for writing data back to disk: write\_csv() and write\_tsv(). Both functions increase the chances of the output file being read back in correctly by:

* Always encoding strings in UTF-8.
* Saving dates and date-times in ISO8601 format so they are easily parsed elsewhere.

If you want to export a csv file to Excel, use write\_excel\_csv() --- this writes a special character (a "byte order mark") at the start of the file which tells Excel that you're using the UTF-8 encoding.

The most important arguments are x (the data frame to save), and path (the location to save it). You can also specify how missing values are written with na, and if you want to append to an existing file.

write\_csv(heights, "results/heights.csv")

Note that the type information is lost when you save to csv:

heights

## # A tibble: 1,192 × 6  
## earn height sex ed age race  
## <dbl> <dbl> <chr> <int> <int> <chr>  
## 1 50000 74.42444 male 16 45 white  
## 2 60000 65.53754 female 16 58 white  
## 3 30000 63.62920 female 16 29 white  
## 4 50000 63.10856 female 16 91 other  
## 5 51000 63.40248 female 17 39 white  
## 6 9000 64.39951 female 15 26 white  
## 7 29000 61.65633 female 12 49 white  
## 8 32000 72.69854 male 17 46 white  
## 9 2000 72.03947 male 15 21 hispanic  
## 10 27000 72.23493 male 12 26 white  
## # ... with 1,182 more rows

write\_csv(heights, "results/heights-2.csv")  
read\_csv("results/heights-2.csv")

## # A tibble: 1,192 × 6  
## earn height sex ed age race  
## <dbl> <dbl> <chr> <int> <int> <chr>  
## 1 50000 74.42444 male 16 45 white  
## 2 60000 65.53754 female 16 58 white  
## 3 30000 63.62920 female 16 29 white  
## 4 50000 63.10856 female 16 91 other  
## 5 51000 63.40248 female 17 39 white  
## 6 9000 64.39951 female 15 26 white  
## 7 29000 61.65633 female 12 49 white  
## 8 32000 72.69854 male 17 46 white  
## 9 2000 72.03947 male 15 21 hispanic  
## 10 27000 72.23493 male 12 26 white  
## # ... with 1,182 more rows

This makes CSVs a little unreliable for caching interim results---you need to recreate the column specification every time you load in. There are two alternatives:

1. write\_rds() and read\_rds() are uniform wrappers around the base functions readRDS() and saveRDS(). These store data in R's custom binary format called RDS:

* write\_rds(heights, "results/heights.rds")  
  read\_rds("results/heights.rds")
* ## # A tibble: 1,192 × 6  
  ## earn height sex ed age race  
  ## <dbl> <dbl> <chr> <int> <int> <chr>  
  ## 1 50000 74.42444 male 16 45 white  
  ## 2 60000 65.53754 female 16 58 white  
  ## 3 30000 63.62920 female 16 29 white  
  ## 4 50000 63.10856 female 16 91 other  
  ## 5 51000 63.40248 female 17 39 white  
  ## 6 9000 64.39951 female 15 26 white  
  ## 7 29000 61.65633 female 12 49 white  
  ## 8 32000 72.69854 male 17 46 white  
  ## 9 2000 72.03947 male 15 21 hispanic  
  ## 10 27000 72.23493 male 12 26 white  
  ## # ... with 1,182 more rows

1. The feather package implements a fast binary file format that can be shared across programming languages:

* install.packages("feather")  
  library(feather)  
  write\_feather(heights, "results/heights.feather")  
  read\_feather("results/heights.feather")

Feather tends to be faster than RDS and is usable outside of R. RDS supports list-columns (which we will come back to in [Model]); feather currently does not.

## Learn More

* [R Data Import Tutorial](https://www.datacamp.com/community/tutorials/r-data-import-tutorial) by Data Camp
* [Importing Data](http://uc-r.github.io/import) and [Exporting Data](http://uc-r.github.io/exporting) by R Programming @ UC.
* [Data Import & Export in R](https://science.nature.nps.gov/im/datamgmt/statistics/r/fundamentals/index.cfm) by National Park Services
* [Scrape web data with APIs](http://stat545.com/cm111_webdata.html)

# Tidy

## Readings

1. [R4DS Chapter 12](http://r4ds.had.co.nz/tidy-data.html)
2. [Wickham, Hadley, 2014. Tidy Data, Journal of Statistical Software Vol 59 (2014), Issue 10, 10.18637/jss.v059.i10](http://www.jstatsoft.org/article/view/v059i10)

## Introduction

In this session, you will learn a consistent way to organise your data in R, an organisation called **tidy data**. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

This session will give you a practical introduction to tidy data and the accompanying tools in the **tidyr** package. If you'd like to learn more about the underlying theory, you might enjoy the *Tidy Data* paper published in the Journal of Statistical Software, <http://www.jstatsoft.org/v59/i10/paper>.

### Prerequisites

In this chapter we'll focus on tidyr, a package that provides a bunch of tools to help tidy up your messy datasets. tidyr is a member of the core tidyverse.

library(tidyverse)

## Tidy data

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organises the values in a different way.

table1

## # A tibble: 6 × 4  
## country year cases population  
## <chr> <int> <int> <int>  
## 1 Afghanistan 1999 745 19987071  
## 2 Afghanistan 2000 2666 20595360  
## 3 Brazil 1999 37737 172006362  
## 4 Brazil 2000 80488 174504898  
## 5 China 1999 212258 1272915272  
## 6 China 2000 213766 1280428583

table2

## # A tibble: 12 × 4  
## country year type count  
## <chr> <int> <chr> <int>  
## 1 Afghanistan 1999 cases 745  
## 2 Afghanistan 1999 population 19987071  
## 3 Afghanistan 2000 cases 2666  
## 4 Afghanistan 2000 population 20595360  
## 5 Brazil 1999 cases 37737  
## 6 Brazil 1999 population 172006362  
## 7 Brazil 2000 cases 80488  
## 8 Brazil 2000 population 174504898  
## 9 China 1999 cases 212258  
## 10 China 1999 population 1272915272  
## 11 China 2000 cases 213766  
## 12 China 2000 population 1280428583

table3

## # A tibble: 6 × 3  
## country year rate  
## \* <chr> <int> <chr>  
## 1 Afghanistan 1999 745/19987071  
## 2 Afghanistan 2000 2666/20595360  
## 3 Brazil 1999 37737/172006362  
## 4 Brazil 2000 80488/174504898  
## 5 China 1999 212258/1272915272  
## 6 China 2000 213766/1280428583

# Spread across two tibbles  
table4a # cases

## # A tibble: 3 × 3  
## country `1999` `2000`  
## \* <chr> <int> <int>  
## 1 Afghanistan 745 2666  
## 2 Brazil 37737 80488  
## 3 China 212258 213766

table4b # population

## # A tibble: 3 × 3  
## country `1999` `2000`  
## \* <chr> <int> <int>  
## 1 Afghanistan 19987071 20595360  
## 2 Brazil 172006362 174504898  
## 3 China 1272915272 1280428583

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

* Each variable must have its own column.
* Each observation must have its own row.
* Each value must have its own cell.



tidy data

These three rules are interrelated because it’s impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

* Put each dataset in a tibble.
* Put each variable in a column.

In this example, only table1 is tidy. It’s the only representation where each column is a variable.

Why ensure that your data is tidy? There are two main advantages:

1. There’s a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it’s easier to learn the tools that work with it because they have an underlying uniformity.
2. There’s a specific advantage to placing variables in columns because it allows R’s vectorised nature to shine. As you learned in mutate and summary functions, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

Why ensure that your data is tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. As you learned in [mutate](#mutate-funs) and [summary functions](#summary-funs), most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with table1.

## Spreading and gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidyr: gather() and spread().

### Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take table4a: the column names 1999 and 2000 represent values of the year variable, and each row represents two observations, not one.

table4a

## # A tibble: 3 × 3  
## country `1999` `2000`  
## \* <chr> <int> <int>  
## 1 Afghanistan 745 2666  
## 2 Brazil 37737 80488  
## 3 China 212258 213766

To tidy a dataset like this, we need to **gather** those columns into a new pair of variables. To describe that operation we need three parameters:

* The set of columns that represent values, not variables. In this example, those are the columns 1999 and 2000.
* The name of the variable whose values form the column names. I call that the key, and here it is year.
* The name of the variable whose values are spread over the cells. I call that value, and here it's the number of cases.

Together those parameters generate the call to gather():

table4a %>%   
 gather(`1999`, `2000`, key = "year", value = "cases")

## # A tibble: 6 × 3  
## country year cases  
## <chr> <chr> <int>  
## 1 Afghanistan 1999 745  
## 2 Brazil 1999 37737  
## 3 China 1999 212258  
## 4 Afghanistan 2000 2666  
## 5 Brazil 2000 80488  
## 6 China 2000 213766

The columns to gather are specified with dplyr::select() style notation. Here there are only two columns, so we list them individually. Note that "1999" and "2000" are non-syntactic names so we have to surround them in backticks. To refresh your memory of the other ways to select columns, see [select](#select).

Figure 2 Gathering table4 into a tidy form.

Figure 2 Gathering table4 into a tidy form.

In the final result, the gathered columns are dropped, and we get new key and value columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure 2. We can use gather() to tidy table4b in a similar fashion. The only difference is the variable stored in the cell values:

table4b %>%   
 gather(`1999`, `2000`, key = "year", value = "population")

## # A tibble: 6 × 3  
## country year population  
## <chr> <chr> <int>  
## 1 Afghanistan 1999 19987071  
## 2 Brazil 1999 172006362  
## 3 China 1999 1272915272  
## 4 Afghanistan 2000 20595360  
## 5 Brazil 2000 174504898  
## 6 China 2000 1280428583

To combine the tidied versions of table4a and table4b into a single tibble, we need to use dplyr::left\_join(), which you'll learn about in [relational data].

tidy4a <- table4a %>%   
 gather(`1999`, `2000`, key = "year", value = "cases")  
tidy4b <- table4b %>%   
 gather(`1999`, `2000`, key = "year", value = "population")  
left\_join(tidy4a, tidy4b)

## # A tibble: 6 × 4  
## country year cases population  
## <chr> <chr> <int> <int>  
## 1 Afghanistan 1999 745 19987071  
## 2 Brazil 1999 37737 172006362  
## 3 China 1999 212258 1272915272  
## 4 Afghanistan 2000 2666 20595360  
## 5 Brazil 2000 80488 174504898  
## 6 China 2000 213766 1280428583

### Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take table2: an observation is a country in a year, but each observation is spread across two rows.

table2

## # A tibble: 12 × 4  
## country year type count  
## <chr> <int> <chr> <int>  
## 1 Afghanistan 1999 cases 745  
## 2 Afghanistan 1999 population 19987071  
## 3 Afghanistan 2000 cases 2666  
## 4 Afghanistan 2000 population 20595360  
## 5 Brazil 1999 cases 37737  
## 6 Brazil 1999 population 172006362  
## 7 Brazil 2000 cases 80488  
## 8 Brazil 2000 population 174504898  
## 9 China 1999 cases 212258  
## 10 China 1999 population 1272915272  
## 11 China 2000 cases 213766  
## 12 China 2000 population 1280428583

To tidy this up, we first analyse the representation in similar way to gather(). This time, however, we only need two parameters:

* The column that contains variable names, the key column. Here, it's type.
* The column that contains values forms multiple variables, the value column. Here it's count.

Once we've figured that out, we can use spread(), as shown programmatically below, and visually in Figure 3.

spread(table2, key = type, value = count)

## # A tibble: 6 × 4  
## country year cases population  
## \* <chr> <int> <int> <int>  
## 1 Afghanistan 1999 745 19987071  
## 2 Afghanistan 2000 2666 20595360  
## 3 Brazil 1999 37737 172006362  
## 4 Brazil 2000 80488 174504898  
## 5 China 1999 212258 1272915272  
## 6 China 2000 213766 1280428583

Figure 3 Spreading table2 makes it tidy

Figure 3 Spreading table2 makes it tidy

As you might have guessed from the common key and value arguments, spread() and gather() are complements. gather() makes wide tables narrower and longer; spread() makes long tables shorter and wider.

### Exercises

1. Why are gather() and spread() not perfectly symmetrical?  
   Carefully consider the following example:

* stocks <- tibble(  
   year = c(2015, 2015, 2016, 2016),  
   half = c( 1, 2, 1, 2),  
   return = c(1.88, 0.59, 0.92, 0.17)  
  )  
  stocks %>%   
   spread(year, return) %>%   
   gather("year", "return", `2015`:`2016`)
* (Hint: look at the variable types and think about column *names*.)
* Both spread() and gather() have a convert argument. What does it do?

1. Why does this code fail?

* table4a %>%   
   gather(1999, 2000, key = "year", value = "cases")
* ## Error in combine\_vars(vars, ind\_list): Position must be between 0 and n

1. Why does spreading this tibble fail? How could you add a new column to fix the problem?

* people <- tribble(  
   ~name, ~key, ~value,  
   #-----------------|--------|------  
   "Phillip Woods", "age", 45,  
   "Phillip Woods", "height", 186,  
   "Phillip Woods", "age", 50,  
   "Jessica Cordero", "age", 37,  
   "Jessica Cordero", "height", 156  
  )

1. Tidy the simple tibble below. Do you need to spread or gather it? What are the variables?

* preg <- tribble(  
   ~pregnant, ~male, ~female,  
   "yes", NA, 10,  
   "no", 20, 12  
  )

## Separating and uniting

So far you've learned how to tidy table2 and table4, but not table3. table3 has a different problem: we have one column (rate) that contains two variables (cases and population). To fix this problem, we'll need the separate() function. You'll also learn about the complement of separate(): unite(), which you use if a single variable is spread across multiple columns.

### Separate

separate() pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take table3:

table3

## # A tibble: 6 × 3  
## country year rate  
## \* <chr> <int> <chr>  
## 1 Afghanistan 1999 745/19987071  
## 2 Afghanistan 2000 2666/20595360  
## 3 Brazil 1999 37737/172006362  
## 4 Brazil 2000 80488/174504898  
## 5 China 1999 212258/1272915272  
## 6 China 2000 213766/1280428583

The rate column contains both cases and population variables, and we need to split it into two variables. separate() takes the name of the column to separate, and the names of the columns to separate into, as shown in Figure 4 and the code below.

table3 %>%   
 separate(rate, into = c("cases", "population"))

## # A tibble: 6 × 4  
## country year cases population  
## \* <chr> <int> <chr> <chr>  
## 1 Afghanistan 1999 745 19987071  
## 2 Afghanistan 2000 2666 20595360  
## 3 Brazil 1999 37737 172006362  
## 4 Brazil 2000 80488 174504898  
## 5 China 1999 212258 1272915272  
## 6 China 2000 213766 1280428583

Figure 4 Separating table3 makes it tidy

Figure 4 Separating table3 makes it tidy

By default, separate() will split values wherever it sees a non-alphanumeric character (i.e. a character that isn't a number or letter). For example, in the code above, separate() split the values of rate at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the sep argument of separate(). For example, we could rewrite the code above as:

table3 %>%   
 separate(rate, into = c("cases", "population"), sep = "/")

(Formally, sep is a regular expression, which you'll learn more about in [strings].)

Look carefully at the column types: you'll notice that case and population are character columns. This is the default behaviour in separate(): it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask separate() to try and convert to better types using convert = TRUE:

table3 %>%   
 separate(rate, into = c("cases", "population"), convert = TRUE)

## # A tibble: 6 × 4  
## country year cases population  
## \* <chr> <int> <int> <int>  
## 1 Afghanistan 1999 745 19987071  
## 2 Afghanistan 2000 2666 20595360  
## 3 Brazil 1999 37737 172006362  
## 4 Brazil 2000 80488 174504898  
## 5 China 1999 212258 1272915272  
## 6 China 2000 213766 1280428583

You can also pass a vector of integers to sep. separate() will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings. When using integers to separate strings, the length of sep should be one less than the number of names in into.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

table3 %>%   
 separate(year, into = c("century", "year"), sep = 2)

## # A tibble: 6 × 4  
## country century year rate  
## \* <chr> <chr> <chr> <chr>  
## 1 Afghanistan 19 99 745/19987071  
## 2 Afghanistan 20 00 2666/20595360  
## 3 Brazil 19 99 37737/172006362  
## 4 Brazil 20 00 80488/174504898  
## 5 China 19 99 212258/1272915272  
## 6 China 20 00 213766/1280428583

### Unite

unite() is the inverse of separate(): it combines multiple columns into a single column. You'll need it much less frequently than separate(), but it's still a useful tool to have in your back pocket.

Figure 5 Uniting table5 makes it tidy

Figure 5 Uniting table5 makes it tidy

We can use unite() to rejoin the *century* and *year* columns that we created in the last example. That data is saved as tidyr::table5. unite() takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in dplyr::select() style:

table5 %>%   
 unite(new, century, year)

## # A tibble: 6 × 3  
## country new rate  
## \* <chr> <chr> <chr>  
## 1 Afghanistan 19\_99 745/19987071  
## 2 Afghanistan 20\_00 2666/20595360  
## 3 Brazil 19\_99 37737/172006362  
## 4 Brazil 20\_00 80488/174504898  
## 5 China 19\_99 212258/1272915272  
## 6 China 20\_00 213766/1280428583

In this case we also need to use the sep argument. The default will place an underscore (\_) between the values from different columns. Here we don't want any separator so we use "":

table5 %>%   
 unite(new, century, year, sep = "")

## # A tibble: 6 × 3  
## country new rate  
## \* <chr> <chr> <chr>  
## 1 Afghanistan 1999 745/19987071  
## 2 Afghanistan 2000 2666/20595360  
## 3 Brazil 1999 37737/172006362  
## 4 Brazil 2000 80488/174504898  
## 5 China 1999 212258/1272915272  
## 6 China 2000 213766/1280428583

## Case Study

To finish off the session, let's revisit the inflamation data we worked with in [Programming with R](#programming-with-r).

* Is the inflamation data tidy data?
* If not, why not? And how can we tidy it?

# Transform with dplyr

## Readings

1. [R4DS Data Transformation](http://r4ds.had.co.nz/transform.html)
2. [R4DS Relational data](http://r4ds.had.co.nz/relational-data.html)

## Prerequisites

Load dplyr (or tidyverse, which will load dplyr) and gapminder (install it if not yet installed):

library(gapminder)  
library(tidyverse)

## dplyr introduction

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 the [tidyverse](https://github.com/hadley/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](http://plyr.had.co.nz), which implements the ["split-apply-combine" strategy for data analysis](https://www.jstatsoft.org/article/view/v040i01) (PDF). 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 treament 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 alot, you might enjoy learning other ways to iterate over rows or groups of rows or variables in a data frame.

### Say hello to the Gapminder tibble

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

gapminder

## # A tibble: 1,704 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414  
## # ... 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 × 5  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## <dbl> <dbl> <dbl> <dbl> <fctr>  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 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 5.0 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 5.0 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

### 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 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Canada Americas 1952 68.750 14785584 11367.16  
## 2 Canada Americas 1957 69.960 17010154 12489.95  
## 3 Canada Americas 1962 71.300 18985849 13462.49  
## 4 Canada Americas 1967 72.130 20819767 16076.59  
## 5 Canada Americas 1972 72.880 22284500 18970.57  
## 6 Canada Americas 1977 74.210 23796400 22090.88  
## 7 Canada Americas 1982 75.760 25201900 22898.79  
## 8 Canada Americas 1987 76.860 26549700 26626.52  
## 9 Canada Americas 1992 77.950 28523502 26342.88  
## 10 Canada Americas 1997 78.610 30305843 28954.93  
## 11 Canada Americas 2002 79.770 31902268 33328.97  
## 12 Canada Americas 2007 80.653 33390141 36319.24

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 facetting 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.

### 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 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Rwanda Africa 1992 23.599 7290203 737.0686

filter(gapminder, country == "Rwanda", year > 1979)

## # A tibble: 6 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Rwanda Africa 1982 46.218 5507565 881.5706  
## 2 Rwanda Africa 1987 44.020 6349365 847.9912  
## 3 Rwanda Africa 1992 23.599 7290203 737.0686  
## 4 Rwanda Africa 1997 36.087 7212583 589.9445  
## 5 Rwanda Africa 2002 43.413 7852401 785.6538  
## 6 Rwanda Africa 2007 46.242 8860588 863.0885

filter(gapminder, country %in% c("Rwanda", "Afghanistan"))

## # A tibble: 24 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414  
## # ... 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.

### Meet the new pipe operator

Before we go any further, we should exploit the new pipe operator that the tidyverse imports from the [magrittr](https://github.com/smbache/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](http://blogue.us/wp-content/uploads/2009/07/Unknown-21.jpeg). 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 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134

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 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007

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

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

### 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 × 2  
## year lifeExp  
## <int> <dbl>  
## 1 1952 28.801  
## 2 1957 30.332  
## 3 1962 31.997  
## 4 1967 34.020  
## 5 1972 36.088  
## 6 1977 38.438  
## 7 1982 39.854  
## 8 1987 40.822  
## 9 1992 41.674  
## 10 1997 41.763  
## # ... 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 × 2  
## year lifeExp  
## <int> <dbl>  
## 1 1952 28.801  
## 2 1957 30.332  
## 3 1962 31.997  
## 4 1967 34.020

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

### Revel in the convenience

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

gapminder %>%  
 filter(country == "Cambodia") %>%  
 select(year, lifeExp)

## # A tibble: 12 × 2  
## year lifeExp  
## <int> <dbl>  
## 1 1952 39.417  
## 2 1957 41.366  
## 3 1962 43.415  
## 4 1967 45.415  
## 5 1972 40.317  
## 6 1977 31.220  
## 7 1982 50.957  
## 8 1987 53.914  
## 9 1992 55.803  
## 10 1997 56.534  
## 11 2002 56.752  
## 12 2007 59.723

and what a typical base R call would look like:

gapminder[gapminder$country == "Cambodia", c("year", "lifeExp")]

## # A tibble: 12 × 2  
## year lifeExp  
## <int> <dbl>  
## 1 1952 39.417  
## 2 1957 41.366  
## 3 1962 43.415  
## 4 1967 45.415  
## 5 1972 40.317  
## 6 1977 31.220  
## 7 1982 50.957  
## 8 1987 53.914  
## 9 1992 55.803  
## 10 1997 56.534  
## 11 2002 56.752  
## 12 2007 59.723

### 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](block010_dplyr-end-single-table.html).

dplyr's verbs, such as filter() and select(), are what's called [pure functions](http://en.wikipedia.org/wiki/Pure_function). To quote from Wickham's [Advanced R Programming book](http://adv-r.had.co.nz/Functions.html):

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](http://adv-r.had.co.nz/dsl.html) for data analysis.

Go to the next block, [dplyr functions for a single dataset](block010_dplyr-end-single-table.html), for more dplyr!

## dplyr grammar

### 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 × 6  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414  
## # ... 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 == "United States")

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

## store the output as an R object  
us <- my\_gap %>% filter(country == "United States")

### 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 × 7  
## country continent year lifeExp pop gdpPercap gdp  
## <fctr> <fctr> <int> <dbl> <int> <dbl> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453 6567086330  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530 7585448670  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007 8758855797  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971 9648014150  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811 9678553274  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134 11697659231  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114 12598563401  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959 11820990309  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414 10595901589  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414 14121995875  
## # ... with 1,694 more rows

Hmmmm ... those GDP numbers are almost uselessly large and abstract. Consider the [advice of Randall Munroe of xkcd](http://fivethirtyeight.com/datalab/xkcd-randall-munroe-qanda-what-if/):

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 a benchmark country.

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:

* Filter down to the rows for Canada
* Create a new temporary variable in my\_gap:
  + Extract the gdpPercap variable from the Canadian data.
  + Replicate it once per country in the dataset, so it has the right length.
* Divide raw gdpPercap by this Canadian figure.
* Discard the temporary variable of replicated Canadian gdpPercap.

ctib <- my\_gap %>%  
 filter(country == "Canada") %>%   
 select(year, BasegdpPercap=gdpPercap)  
ctib

## # A tibble: 12 × 2  
## year BasegdpPercap  
## <int> <dbl>  
## 1 1952 11367.16  
## 2 1957 12489.95  
## 3 1962 13462.49  
## 4 1967 16076.59  
## 5 1972 18970.57  
## 6 1977 22090.88  
## 7 1982 22898.79  
## 8 1987 26626.52  
## 9 1992 26342.88  
## 10 1997 28954.93  
## 11 2002 33328.97  
## 12 2007 36319.24

my\_gap <- my\_gap %>%  
 inner\_join(ctib, by="year") %>%   
 mutate(gdpPercapRel = gdpPercap / BasegdpPercap,  
 BasegdpPercap = NULL)  
my\_gap

## # A tibble: 1,704 × 7  
## country continent year lifeExp pop gdpPercap gdpPercapRel  
## <fctr> <fctr> <int> <dbl> <int> <dbl> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453 0.06856992  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530 0.06572108  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007 0.06336874  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971 0.05201335  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811 0.03900679  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134 0.03558542  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114 0.04271018  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959 0.03201305  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414 0.02464959  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414 0.02194243  
## # ... with 1,694 more rows

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 × 3  
## country year gdpPercapRel  
## <fctr> <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.   
## 0.007236 0.061650 0.171500 0.326700 0.446600 9.535000

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.

### 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 × 7  
## country continent year lifeExp pop gdpPercap gdpPercapRel  
## <fctr> <fctr> <int> <dbl> <int> <dbl> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453 0.06856992  
## 2 Albania Europe 1952 55.230 1282697 1601.0561 0.14084925  
## 3 Algeria Africa 1952 43.077 9279525 2449.0082 0.21544589  
## 4 Angola Africa 1952 30.015 4232095 3520.6103 0.30971764  
## 5 Argentina Americas 1952 62.485 17876956 5911.3151 0.52003442  
## 6 Australia Oceania 1952 69.120 8691212 10039.5956 0.88321046  
## 7 Austria Europe 1952 66.800 6927772 6137.0765 0.53989527  
## 8 Bahrain Asia 1952 50.939 120447 9867.0848 0.86803421  
## 9 Bangladesh Asia 1952 37.484 46886859 684.2442 0.06019482  
## 10 Belgium Europe 1952 68.000 8730405 8343.1051 0.73396559  
## # ... 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 × 7  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Swaziland Africa 2007 39.613 1133066 4513.4806  
## 2 Mozambique Africa 2007 42.082 19951656 823.6856  
## 3 Zambia Africa 2007 42.384 11746035 1271.2116  
## 4 Sierra Leone Africa 2007 42.568 6144562 862.5408  
## 5 Lesotho Africa 2007 42.592 2012649 1569.3314  
## 6 Angola Africa 2007 42.731 12420476 4797.2313  
## 7 Zimbabwe Africa 2007 43.487 12311143 469.7093  
## 8 Afghanistan Asia 2007 43.828 31889923 974.5803  
## 9 Central African Republic Africa 2007 44.741 4369038 706.0165  
## 10 Liberia Africa 2007 45.678 3193942 414.5073  
## # ... with 132 more rows, and 1 more variables: gdpPercapRel <dbl>

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 × 7  
## country continent year lifeExp pop gdpPercap  
## <fctr> <fctr> <int> <dbl> <int> <dbl>  
## 1 Japan Asia 2007 82.603 127467972 31656.07  
## 2 Hong Kong, China Asia 2007 82.208 6980412 39724.98  
## 3 Iceland Europe 2007 81.757 301931 36180.79  
## 4 Switzerland Europe 2007 81.701 7554661 37506.42  
## 5 Australia Oceania 2007 81.235 20434176 34435.37  
## 6 Spain Europe 2007 80.941 40448191 28821.06  
## 7 Sweden Europe 2007 80.884 9031088 33859.75  
## 8 Israel Asia 2007 80.745 6426679 25523.28  
## 9 France Europe 2007 80.657 61083916 30470.02  
## 10 Canada Americas 2007 80.653 33390141 36319.24  
## # ... with 132 more rows, and 1 more variables: gdpPercapRel <dbl>

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.

### Use rename() to rename variables

The Gapminder use [camelCase](http://en.wikipedia.org/wiki/CamelCase) for variable naming, an alternative convention is [snake\_case](http://en.wikipedia.org/wiki/Snake_case). Let's rename some variables to use snake case!

my\_gap %>%  
 rename(life\_exp = lifeExp,  
 gdp\_percap = gdpPercap,  
 gdp\_percap\_rel = gdpPercapRel)

## # A tibble: 1,704 × 7  
## country continent year life\_exp pop gdp\_percap gdp\_percap\_rel  
## <fctr> <fctr> <int> <dbl> <int> <dbl> <dbl>  
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453 0.06856992  
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530 0.06572108  
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007 0.06336874  
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971 0.05201335  
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811 0.03900679  
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134 0.03558542  
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114 0.04271018  
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959 0.03201305  
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414 0.02464959  
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414 0.02194243  
## # ... 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.

### 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 × 3  
## gdpPercap yr lifeExp  
## <dbl> <int> <dbl>  
## 1 463.1151 1997 45.326  
## 2 446.4035 2002 47.360  
## 3 430.0707 2007 49.580

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

### group\_by() is a mighty weapon

I have found ~~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 observations, computes requested summaries, and returns a dataset with 1 observation.
* Window functions take a dataset with observations and return a dataset with 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.

#### 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())

## # A tibble: 5 × 2  
## continent n  
## <fctr> <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 × 2  
## continent n  
## <fctr> <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 × 2  
## continent n  
## <fctr> <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))

## # A tibble: 5 × 3  
## continent n n\_countries  
## <fctr> <int> <int>  
## 1 Africa 624 52  
## 2 Americas 300 25  
## 3 Asia 396 33  
## 4 Europe 360 30  
## 5 Oceania 24 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 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))

## # A tibble: 5 × 2  
## continent avg\_lifeExp  
## <fctr> <dbl>  
## 1 Africa 48.86533  
## 2 Americas 64.65874  
## 3 Asia 60.06490  
## 4 Europe 71.90369  
## 5 Oceania 74.32621

summarize\_each() 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) %>%  
 summarise\_each(funs(mean, median), lifeExp, gdpPercap)

## Source: local data frame [10 x 6]  
## Groups: continent [?]  
##   
## continent year lifeExp\_mean gdpPercap\_mean lifeExp\_median  
## <fctr> <int> <dbl> <dbl> <dbl>  
## 1 Africa 1952 39.13550 1252.572 38.8330  
## 2 Africa 2007 54.80604 3089.033 52.9265  
## 3 Americas 1952 53.27984 4079.063 54.7450  
## 4 Americas 2007 73.60812 11003.032 72.8990  
## 5 Asia 1952 46.31439 5195.484 44.8690  
## 6 Asia 2007 70.72848 12473.027 72.3960  
## 7 Europe 1952 64.40850 5661.057 65.9000  
## 8 Europe 2007 77.64860 25054.482 78.6085  
## 9 Oceania 1952 69.25500 10298.086 69.2550  
## 10 Oceania 2007 80.71950 29810.188 80.7195  
## # ... with 1 more variables: 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))

## # A tibble: 12 × 3  
## year min\_lifeExp max\_lifeExp  
## <int> <dbl> <dbl>  
## 1 1952 28.801 65.390  
## 2 1957 30.332 67.840  
## 3 1962 31.997 69.390  
## 4 1967 34.020 71.430  
## 5 1972 36.088 73.420  
## 6 1977 31.220 75.380  
## 7 1982 39.854 77.110  
## 8 1987 40.822 78.670  
## 9 1992 41.674 79.360  
## 10 1997 41.763 80.690  
## 11 2002 42.129 82.000  
## 12 2007 43.828 82.603

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.

### Grouped mutate

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

#### 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)

## Source: local data frame [426 x 4]  
## Groups: country [142]  
##   
## country year lifeExp lifeExp\_gain  
## <fctr> <int> <dbl> <dbl>  
## 1 Afghanistan 1952 28.801 0.000  
## 2 Afghanistan 1957 30.332 1.531  
## 3 Afghanistan 1962 31.997 3.196  
## 4 Albania 1952 55.230 0.000  
## 5 Albania 1957 59.280 4.050  
## 6 Albania 1962 64.820 9.590  
## 7 Algeria 1952 43.077 0.000  
## 8 Algeria 1957 45.685 2.608  
## 9 Algeria 1962 48.303 5.226  
## 10 Angola 1952 30.015 0.000  
## # ... with 416 more rows

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

#### Window functions

Window functions take inputs and give back 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)

## Source: local data frame [24 x 3]  
## Groups: year [12]  
##   
## year country lifeExp  
## <int> <fctr> <dbl>  
## 1 1952 Afghanistan 28.801  
## 2 1952 Israel 65.390  
## 3 1957 Afghanistan 30.332  
## 4 1957 Israel 67.840  
## 5 1962 Afghanistan 31.997  
## 6 1962 Israel 69.390  
## 7 1967 Afghanistan 34.020  
## 8 1967 Japan 71.430  
## 9 1972 Afghanistan 36.088  
## 10 1972 Japan 73.420  
## 11 1977 Cambodia 31.220  
## 12 1977 Japan 75.380  
## 13 1982 Afghanistan 39.854  
## 14 1982 Japan 77.110  
## 15 1987 Afghanistan 40.822  
## 16 1987 Japan 78.670  
## 17 1992 Afghanistan 41.674  
## 18 1992 Japan 79.360  
## 19 1997 Afghanistan 41.763  
## 20 1997 Japan 80.690  
## 21 2002 Afghanistan 42.129  
## 22 2002 Japan 82.000  
## 23 2007 Afghanistan 43.828  
## 24 2007 Japan 82.603

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

## Source: local data frame [396 x 3]  
## Groups: year [12]  
##   
## year country lifeExp  
## <int> <fctr> <dbl>  
## 1 1952 Afghanistan 28.801  
## 2 1957 Afghanistan 30.332  
## 3 1962 Afghanistan 31.997  
## 4 1967 Afghanistan 34.020  
## 5 1972 Afghanistan 36.088  
## 6 1977 Afghanistan 38.438  
## 7 1982 Afghanistan 39.854  
## 8 1987 Afghanistan 40.822  
## 9 1992 Afghanistan 41.674  
## 10 1997 Afghanistan 41.763  
## # ... 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)

## Source: local data frame [9 x 5]  
## Groups: year [3]  
##   
## year country lifeExp le\_rank le\_desc\_rank  
## <int> <fctr> <dbl> <int> <int>  
## 1 1997 Afghanistan 41.763 1 33  
## 2 2002 Afghanistan 42.129 1 33  
## 3 2007 Afghanistan 43.828 1 33  
## 4 1997 Japan 80.690 33 1  
## 5 2002 Japan 82.000 33 1  
## 6 2007 Japan 82.603 33 1  
## 7 1997 Thailand 67.521 12 22  
## 8 2002 Thailand 68.564 12 22  
## 9 2007 Thailand 70.616 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

## Source: local data frame [12 x 3]  
## Groups: year [12]  
##   
## year country lifeExp  
## <int> <fctr> <dbl>  
## 1 1952 Afghanistan 28.801  
## 2 1957 Afghanistan 30.332  
## 3 1962 Afghanistan 31.997  
## 4 1967 Afghanistan 34.020  
## 5 1972 Afghanistan 36.088  
## 6 1977 Cambodia 31.220  
## 7 1982 Afghanistan 39.854  
## 8 1987 Afghanistan 40.822  
## 9 1992 Afghanistan 41.674  
## 10 1997 Afghanistan 41.763  
## 11 2002 Afghanistan 42.129  
## 12 2007 Afghanistan 43.828

### 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)

## Source: local data frame [5 x 3]  
## Groups: continent [5]  
##   
## continent country worst\_le\_delta  
## <fctr> <fctr> <dbl>  
## 1 Africa Rwanda -20.421  
## 2 Asia Cambodia -9.097  
## 3 Americas El Salvador -1.511  
## 4 Europe Montenegro -1.464  
## 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](http://tinyurl.com/athenaforehead) -- 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.

## Working with two datasets

When working with two or more datasets, we routinely join them. dplyr adopts the SQL-dialect for joining datasets. Below are examples for those of us who don't speak SQL so good. There are lots of [Venn diagrams re: SQL joins on the interwebs](https://encrypted.google.com/search?q=sql+join&tbm=isch), but I wanted R examples.

[Full documentation](http://www.rdocumentation.org/packages/dplyr) for the dplyr package, which is developed by Hadley Wickham and Romain Francois on [GitHub](https://github.com/hadley/dplyr). The [vignette on Two-table verbs](https://cran.r-project.org/web/packages/dplyr/vignettes/two-table.html) covers the joins shown here.

Working with two small data.frames, superheroes and publishers.

suppressPackageStartupMessages(library(dplyr))  
library(readr)  
  
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  
"  
superheroes <- read\_csv(superheroes, trim\_ws = TRUE, skip = 1)  
  
publishers <- "  
 publisher, yr\_founded  
 DC, 1934  
 Marvel, 1939  
 Image, 1992  
"  
publishers <- read\_csv(publishers, trim\_ws = TRUE, skip = 1)

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.

You probably don't need to memorize each of these functions, but you should be aware of their difference. When coming across problems requiring dataset join, think about which type of joins you need and ?dplyr::join to look up the appropriate function.

### 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))

## # A tibble: 6 × 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 |

### 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))

## # A tibble: 6 × 4  
## name alignment gender publisher  
## <chr> <chr> <chr> <chr>  
## 1 Batman good male DC  
## 2 Joker bad male DC  
## 3 Catwoman bad female DC  
## 4 Magneto bad male Marvel  
## 5 Storm good female Marvel  
## 6 Mystique bad female Marvel

We get a similar result as with inner\_join() but the join result contains only the variables originally found in x = superheroes. But note the row order has changed.

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 |
| Batman | good | male | DC |
| Joker | bad | male | DC |
| Catwoman | bad | female | DC |
| Magneto | bad | male | Marvel |
| Storm | good | female | Marvel |
| Mystique | bad | female | Marvel |

### 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))

## # A tibble: 7 × 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 |

### 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))

## # A tibble: 1 × 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 |

### 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))

## # A tibble: 6 × 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 |

### 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))

## # A tibble: 2 × 2  
## publisher yr\_founded  
## <chr> <int>  
## 1 Marvel 1939  
## 2 DC 1934

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 |
| Marvel | 1939 |
| DC | 1934 |

### 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))

## # A tibble: 7 × 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 NAs 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 |

### 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))

## # A tibble: 1 × 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 |

### 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))

## # A tibble: 8 × 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 NAs 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 |

## Learn more

* package home [on CRAN](http://cran.r-project.org/web/packages/dplyr/index.html)
  + note there are several vignettes, with the [introduction](http://cran.r-project.org/web/packages/dplyr/vignettes/introduction.html) being the most relevant right now
  + the [one on window functions](http://cran.rstudio.com/web/packages/dplyr/vignettes/window-functions.html) will also be interesting to you now
* [RStudio Data Wrangling cheatsheet](https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf), covering dplyr and tidyr. Remember you can get to these via *Help > Cheatsheets.*
* [Excellent slides](https://github.com/tjmahr/MadR_Pipelines) 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](http://www.dataschool.io/dplyr-tutorial-for-faster-data-manipulation-in-r/) by Data School, that includes a link to an R Markdown document and links to videos

# Visualize

## Readings

1. [R4DS Data Visualization](http://r4ds.had.co.nz/data-visualisation.html)
2. [R4DS Exploratory Data Analysis](http://r4ds.had.co.nz/exploratory-data-analysis.html)

## Introduction

This chapter will teach you how to visualise your data using ggplot2. R has several systems for making graphs, but ggplot2 is one of the most elegant and most versatile. ggplot2 implements the **grammar of graphics**, a coherent system for describing and building graphs. With ggplot2, you can do more faster by learning one system and applying it in many places.

If you'd like to learn more about the theoretical underpinnings of ggplot2 before you start, I'd recommend reading "The Layered Grammar of Graphics", <http://vita.had.co.nz/papers/layered-grammar.pdf>.

### Prerequisites

This chapter focusses on ggplot2, one of the core members of the tidyverse. To access the datasets, help pages, and functions that we will use in this chapter, load the tidyverse by running this code:

library(tidyverse)

That one line of code loads the core tidyverse; packages which you will use in almost every data analysis. It also tells you which functions from the tidyverse conflict with functions in base R (or from other packages you might have loaded).

If you run this code and get the error message "there is no package called ‘tidyverse’", you'll need to first install it, then run library() once again.

install.packages("tidyverse")  
library(tidyverse)

You only need to install a package once, but you need to reload it every time you start a new session.

If we need to be explicit about where a function (or dataset) comes from, we'll use the special form package::function(). For example, ggplot2::ggplot() tells you explicitly that we're using the ggplot() function from the ggplot2 package.

## First steps

Let's use our first graph to answer a question: Do cars with big engines use more fuel than cars with small engines? You probably already have an answer, but try to make your answer precise. What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

### The mpg data frame

You can test your answer with the mpg **data frame** found in ggplot2 (aka ggplot2::mpg). A data frame is a rectangular collection of variables (in the columns) and observations (in the rows). mpg contains observations collected by the US Environment Protection Agency on 38 models of cars.

mpg

## # A tibble: 234 × 11  
## manufacturer model displ year cyl trans drv cty hwy  
## <chr> <chr> <dbl> <int> <int> <chr> <chr> <int> <int>  
## 1 audi a4 1.8 1999 4 auto(l5) f 18 29  
## 2 audi a4 1.8 1999 4 manual(m5) f 21 29  
## 3 audi a4 2.0 2008 4 manual(m6) f 20 31  
## 4 audi a4 2.0 2008 4 auto(av) f 21 30  
## 5 audi a4 2.8 1999 6 auto(l5) f 16 26  
## 6 audi a4 2.8 1999 6 manual(m5) f 18 26  
## 7 audi a4 3.1 2008 6 auto(av) f 18 27  
## 8 audi a4 quattro 1.8 1999 4 manual(m5) 4 18 26  
## 9 audi a4 quattro 1.8 1999 4 auto(l5) 4 16 25  
## 10 audi a4 quattro 2.0 2008 4 manual(m6) 4 20 28  
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>

Among the variables in mpg are:

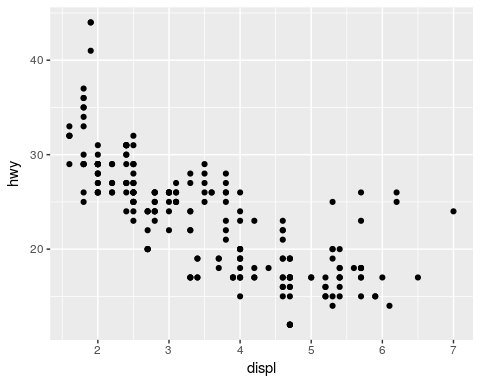
1. displ, a car's engine size, in litres.
2. hwy, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

To learn more about mpg, open its help page by running ?mpg.

### Creating a ggplot

To plot mpg, run this code to put displ on the x-axis and hwy on the y-axis:

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy))



The plot shows a negative relationship between engine size (displ) and fuel efficiency (hwy). In other words, cars with big engines use more fuel. Does this confirm or refute your hypothesis about fuel efficiency and engine size?

With ggplot2, you begin a plot with the function ggplot(). ggplot() creates a coordinate system that you can add layers to. The first argument of ggplot() is the dataset to use in the graph. So ggplot(data = mpg) creates an empty graph, but it's not very interesting so I'm not going to show it here.

You complete your graph by adding one or more layers to ggplot(). The function geom\_point() adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions that each add a different type of layer to a plot. You'll learn a whole bunch of them throughout this chapter.

Each geom function in ggplot2 takes a mapping argument. This defines how variables in your dataset are mapped to visual properties. The mapping argument is always paired with aes(), and the x and y arguments of aes() specify which variables to map to the x and y axes. ggplot2 looks for the mapped variable in the data argument, in this case, mpg.

### A graphing template

Let's turn this code into a reusable template for making graphs with ggplot2. To make a graph, replace the bracketed sections in the code below with a dataset, a geom function, or a collection of mappings.

ggplot(data = <DATA>) +   
 <GEOM\_FUNCTION>(mapping = aes(<MAPPINGS>))

The rest of this chapter will show you how to complete and extend this template to make different types of graphs. We will begin with the <MAPPINGS> component.

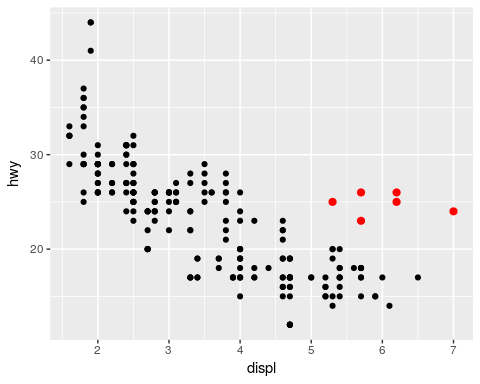
### Exercises

1. Run ggplot(data = mpg) what do you see?
2. How many rows are in mtcars? How many columns?
3. What does the drv variable describe? Read the help for ?mpg to find out.
4. Make a scatterplot of hwy vs cyl.
5. What happens if you make a scatterplot of class vs drv. Why is the plot not useful?

## Aesthetic mappings

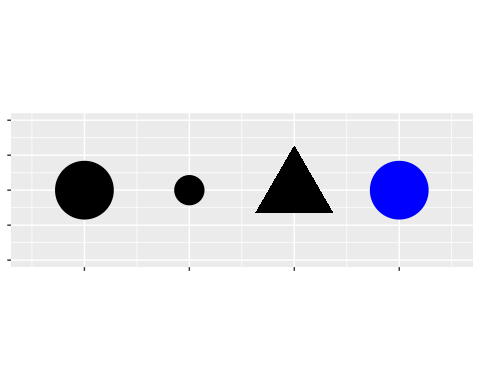
"The greatest value of a picture is when it forces us to notice what we never expected to see." --- John Tukey

In the plot below, one group of points (highlighted in red) seems to fall outside of the linear trend. These cars have a higher mileage than you might expect. How can you explain these cars?



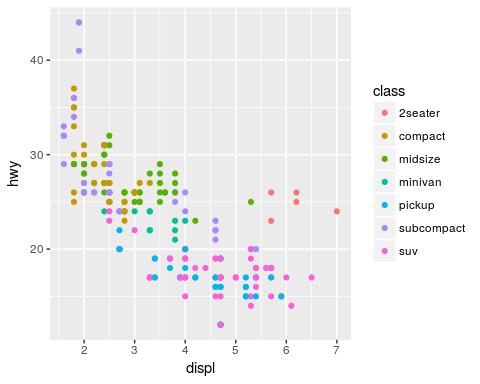
Let's hypothesize that the cars are hybrids. One way to test this hypothesis is to look at the class value for each car. The class variable of the mpg dataset classifies cars into groups such as compact, midsize, and SUV. If the outlying points are hybrids, they should be classified as compact cars or, perhaps, subcompact cars (keep in mind that this data was collected before hybrid trucks and SUVs became popular).

You can add a third variable, like class, to a two dimensional scatterplot by mapping it to an **aesthetic**. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one below) in different ways by changing the values of its aesthetic properties. Since we already use the word "value" to describe data, let's use the word "level" to describe aesthetic properties. Here we change the levels of a point's size, shape, and color to make the point small, triangular, or blue:



You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset. For example, you can map the colors of your points to the class variable to reveal the class of each car.

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy, color = class))



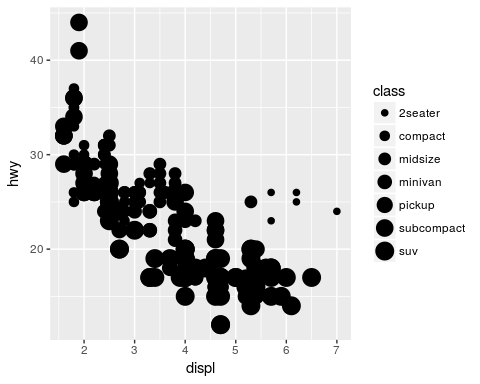
(If you prefer British English, like Hadley, you can use colour instead of color.)

To map an aesthetic to a variable, associate the name of the aesthetic to the name of the variable inside aes(). ggplot2 will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as **scaling**. ggplot2 will also add a legend that explains which levels correspond to which values.

The colors reveal that many of the unusual points are two-seater cars. These cars don't seem like hybrids, and are, in fact, sports cars! Sports cars have large engines like SUVs and pickup trucks, but small bodies like midsize and compact cars, which improves their gas mileage. In hindsight, these cars were unlikely to be hybrids since they have large engines.

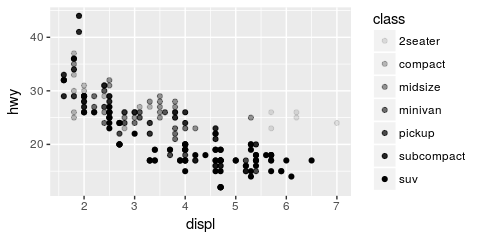
In the above example, we mapped class to the color aesthetic, but we could have mapped class to the size aesthetic in the same way. In this case, the exact size of each point would reveal its class affiliation. We get a *warning* here, because mapping an unordered variable (class) to an ordered aesthetic (size) is not a good idea.

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy, size = class))

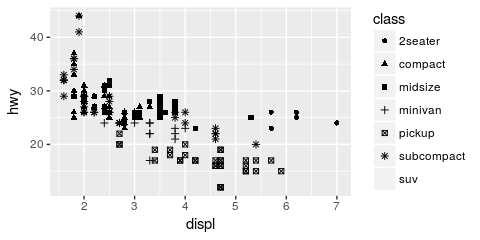


Or we could have mapped class to the *alpha* aesthetic, which controls the transparency of the points, or the shape of the points.

# Left  
ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy, alpha = class))



# Right  
ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy, shape = class))



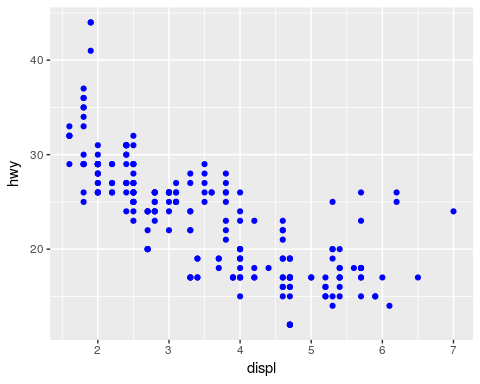
What happened to the SUVs? ggplot2 will only use six shapes at a time. By default, additional groups will go unplotted when you use the shape aesthetic.

For each aesthetic, you use aes() to associate the name of the aesthetic with a variable to display. The aes() function gathers together each of the aesthetic mappings used by a layer and passes them to the layer's mapping argument. The syntax highlights a useful insight about x and y: the x and y locations of a point are themselves aesthetics, visual properties that you can map to variables to display information about the data.

Once you map an aesthetic, ggplot2 takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a legend that explains the mapping between levels and values. For x and y aesthetics, ggplot2 does not create a legend, but it creates an axis line with tick marks and a label. The axis line acts as a legend; it explains the mapping between locations and values.

You can also *set* the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy), color = "blue")



Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function; i.e. it goes *outside* of aes(). You'll need to pick a value that makes sense for that aesthetic:

* The name of a color as a character string.
* The size of a point in mm.
* The shape of a point as a number, as shown in Figure 6.

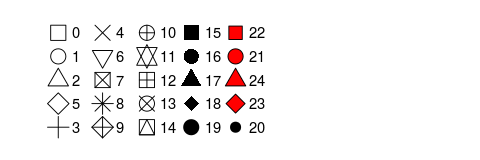
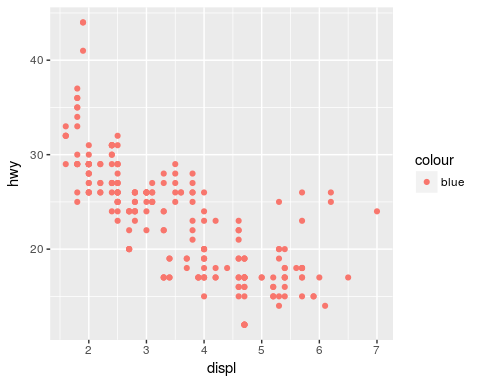


Figure 6 R has 25 built in shapes that are identified by numbers. There are some seeming duplicates: for example, 0, 15, and 22 are all squares. The difference comes from the interaction of the colour and fill aesthetics. The hollow shapes (0--14) have a border determined by colour; the solid shapes (15--18) are filled with colour; the filled shapes (21--24) have a border of colour and are filled with fill.

### Exercises

1. What's gone wrong with this code? Why are the points not blue?

* ggplot(data = mpg) +   
   geom\_point(mapping = aes(x = displ, y = hwy, color = "blue"))
* 

1. Which variables in mpg are categorical? Which variables are continuous? (Hint: type ?mpg to read the documentation for the dataset). How can you see this information when you run mpg?
2. Map a continuous variable to color, size, and shape. How do these aesthetics behave differently for categorical vs. continuous variables?
3. What happens if you map the same variable to multiple aesthetics?
4. What does the stroke aesthetic do? What shapes does it work with? (Hint: use ?geom\_point)
5. What happens if you map an aesthetic to something other than a variable name, like aes(colour = displ < 5)?

## Common problems

As you start to run R code, you're likely to run into problems. Don't worry --- it happens to everyone. I have been writing R code for years, and every day I still write code that doesn't work!

Start by carefully comparing the code that you're running to the code in the book. R is extremely picky, and a misplaced character can make all the difference. Make sure that every ( is matched with a ) and every " is paired with another ". Sometimes you'll run the code and nothing happens. Check the left-hand of your console: if it's a +, it means that R doesn't think you've typed a complete expression and it's waiting for you to finish it. In this case, it's usually easy to start from scratch again by pressing ESCAPE to abort processing the current command.

One common problem when creating ggplot2 graphics is to put the + in the wrong place: it has to come at the end of the line, not the start. In other words, make sure you haven't accidentally written code like this:

ggplot(data = mpg)   
+ geom\_point(mapping = aes(x = displ, y = hwy))

If you're still stuck, try the help. You can get help about any R function by running ?function\_name in the console, or selecting the function name and pressing F1 in RStudio. Don't worry if the help doesn't seem that helpful - instead skip down to the examples and look for code that matches what you're trying to do.

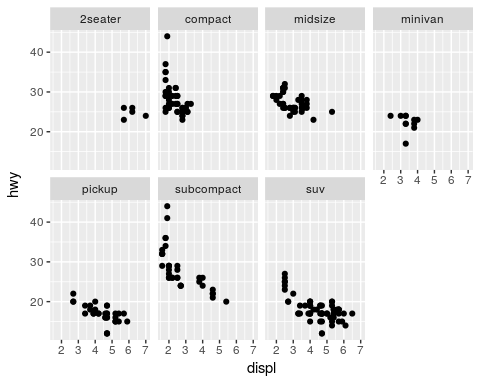
If that doesn't help, carefully read the error message. Sometimes the answer will be buried there! But when you're new to R, the answer might be in the error message but you don't yet know how to understand it. Another great tool is Google: trying googling the error message, as it's likely someone else has had the same problem, and has gotten help online.

## Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into **facets**, subplots that each display one subset of the data.

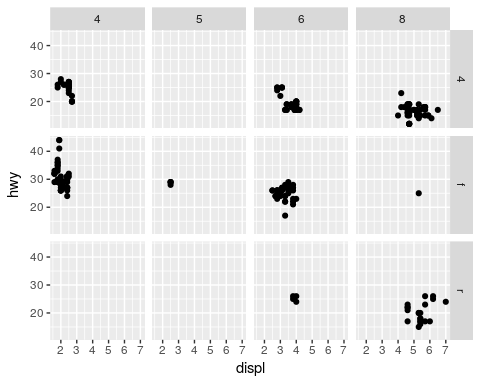
To facet your plot by a single variable, use facet\_wrap(). The first argument of facet\_wrap() should be a formula, which you create with ~ followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation"). The variable that you pass to facet\_wrap() should be discrete.

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy)) +   
 facet\_wrap(~ class, nrow = 2)



To facet your plot on the combination of two variables, add facet\_grid() to your plot call. The first argument of facet\_grid() is also a formula. This time the formula should contain two variable names separated by a ~.

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy)) +   
 facet\_grid(drv ~ cyl)



If you prefer to not facet in the rows or columns dimension, use a . instead of a variable name, e.g. + facet\_grid(. ~ cyl).

### Exercises

1. What happens if you facet on a continuous variable?
2. What do the empty cells in plot with facet\_grid(drv ~ cyl) mean? How do they relate to this plot?

* ggplot(data = mpg) +   
   geom\_point(mapping = aes(x = drv, y = cyl))

1. What plots does the following code make? What does . do?

* ggplot(data = mpg) +   
   geom\_point(mapping = aes(x = displ, y = hwy)) +  
   facet\_grid(drv ~ .)  
    
  ggplot(data = mpg) +   
   geom\_point(mapping = aes(x = displ, y = hwy)) +  
   facet\_grid(. ~ cyl)

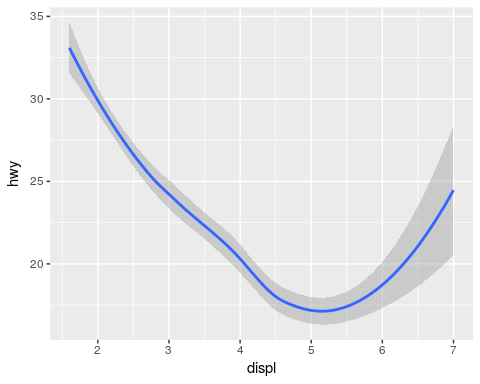
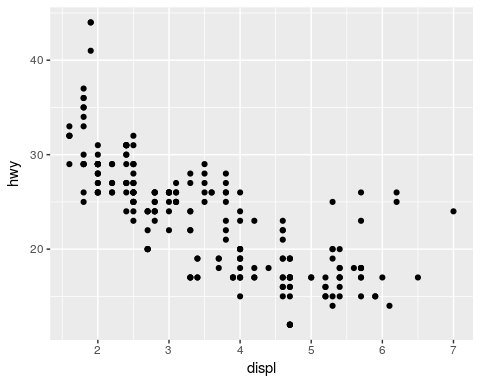
1. Take the first faceted plot in this section:

* ggplot(data = mpg) +   
   geom\_point(mapping = aes(x = displ, y = hwy)) +   
   facet\_wrap(~ class, nrow = 2)
* What are the advantages to using faceting instead of the colour aesthetic? What are the disadvantages? How might the balance change if you had a larger dataset?

1. Read ?facet\_wrap. What does nrow do? What does ncol do? What other options control the layout of the individual panels? Why doesn't facet\_grid() have nrow and ncol variables?
2. When using facet\_grid() you should usually put the variable with more unique levels in the columns. Why?

## Geometric objects

How are these two plots similar?



Both plots contain the same x variable, the same y variable, and both describe the same data. But the plots are not identical. Each plot uses a different visual object to represent the data. In ggplot2 syntax, we say that they use different **geoms**.

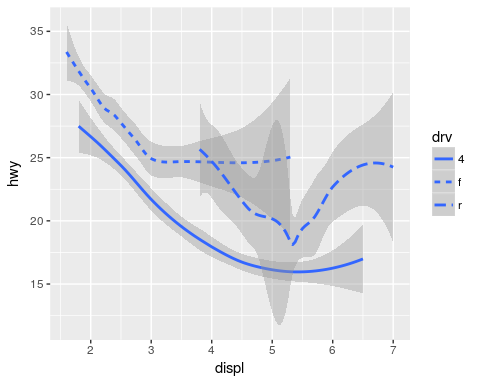
A **geom** is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. As we see above, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to ggplot(). For instance, to make the plots above, you can use this code:

# left  
ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy))  
  
# right  
ggplot(data = mpg) +   
 geom\_smooth(mapping = aes(x = displ, y = hwy))

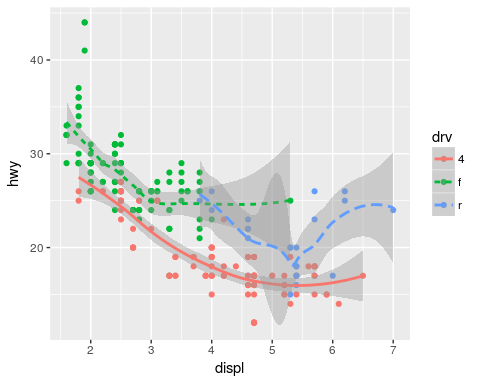
Every geom function in ggplot2 takes a mapping argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn't set the "shape" of a line. On the other hand, you *could* set the linetype of a line. geom\_smooth() will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype.

ggplot(data = mpg) +   
 geom\_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))



Here geom\_smooth() separates the cars into three lines based on their drv value, which describes a car's drivetrain. One line describes all of the points with a 4 value, one line describes all of the points with an f value, and one line describes all of the points with an r value. Here, 4 stands for four-wheel drive, f for front-wheel drive, and r for rear-wheel drive.

If this sounds strange, we can make it more clear by overlaying the lines on top of the raw data and then coloring everything according to drv.

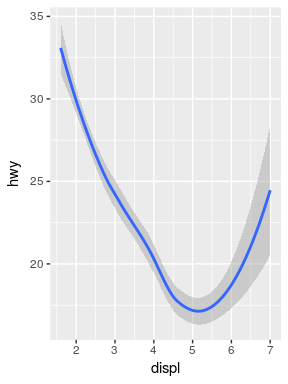


Notice that this plot contains two geoms in the same graph! If this makes you excited, buckle up. In the next section, we will learn how to place multiple geoms in the same plot.

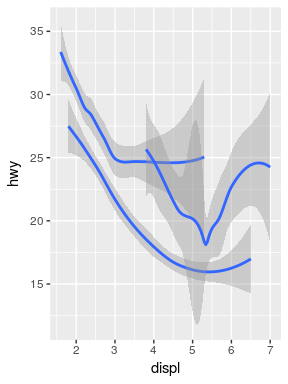
ggplot2 provides over 30 geoms, and extension packages provide even more (see <https://www.ggplot2-exts.org> for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at <http://rstudio.com/cheatsheets>. To learn more about any single geom, use help: ?geom\_smooth.

Many geoms, like geom\_smooth(), use a single geometric object to display multiple rows of data. For these geoms, you can set the group aesthetic to a categorical variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable. In practice, ggplot2 will automatically group the data for these geoms whenever you map an aesthetic to a discrete variable (as in the linetype example). It is convenient to rely on this feature because the group aesthetic by itself does not add a legend or distinguishing features to the geoms.

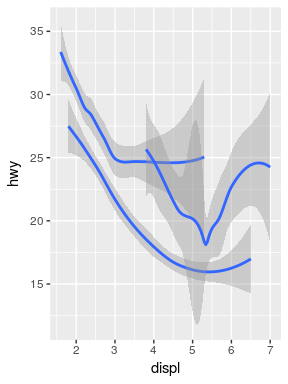
ggplot(data = mpg) +  
 geom\_smooth(mapping = aes(x = displ, y = hwy))



ggplot(data = mpg) +  
 geom\_smooth(mapping = aes(x = displ, y = hwy, group = drv))

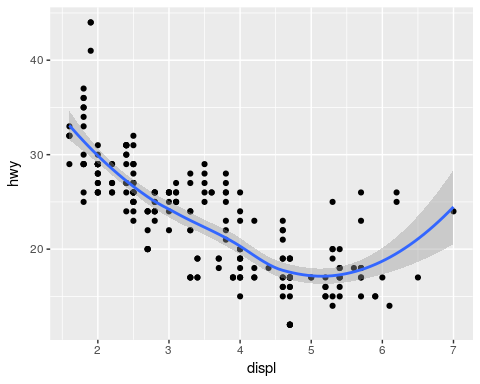


ggplot(data = mpg) +  
 geom\_smooth(  
 mapping = aes(x = displ, y = hwy, group = drv)  
 )



To display multiple geoms in the same plot, add multiple geom functions to ggplot():

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy)) +  
 geom\_smooth(mapping = aes(x = displ, y = hwy))

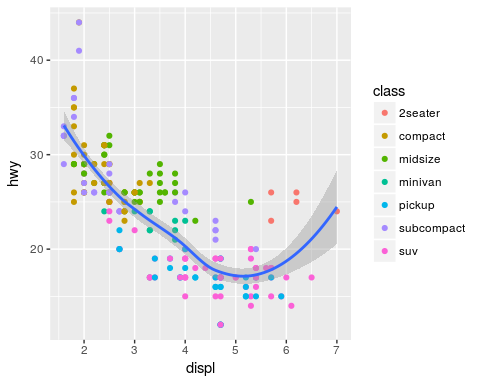


This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display cty instead of hwy. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to ggplot(). ggplot2 will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +   
 geom\_point() +   
 geom\_smooth()

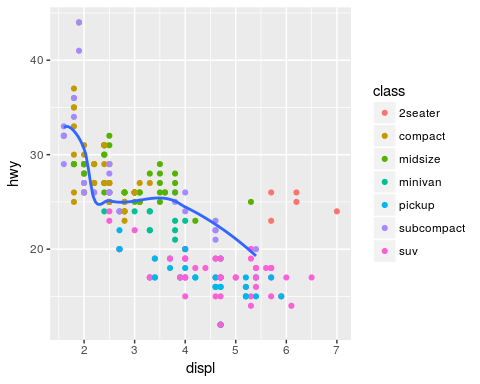
If you place mappings in a geom function, ggplot2 will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers.

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +   
 geom\_point(mapping = aes(color = class)) +   
 geom\_smooth()



You can use the same idea to specify different data for each layer. Here, our smooth line displays just a subset of the mpg dataset, the subcompact cars. The local data argument in geom\_smooth() overrides the global data argument in ggplot() for that layer only.

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +   
 geom\_point(mapping = aes(color = class)) +   
 geom\_smooth(data = filter(mpg, class == "subcompact"), se = FALSE)



(You'll learn how filter() works in the next chapter: for now, just know that this command selects only the subcompact cars.)

### Exercises

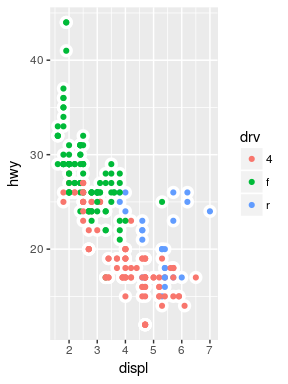
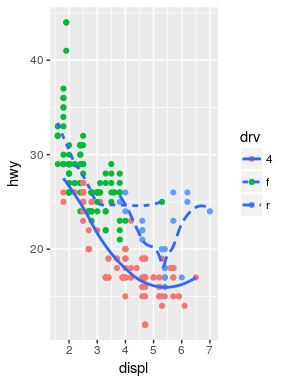
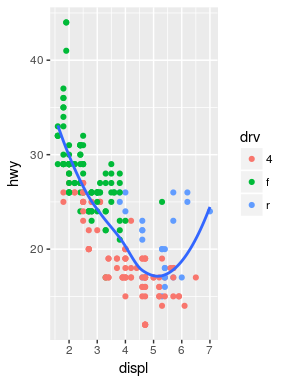
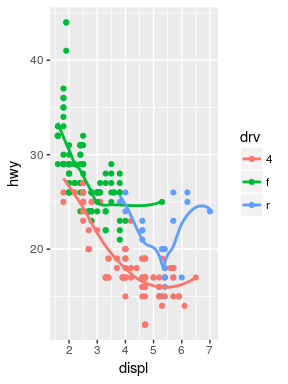
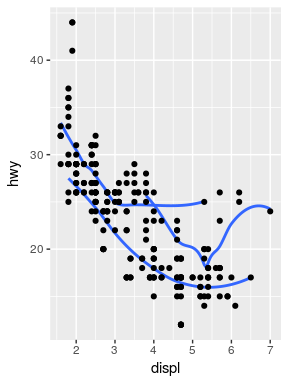
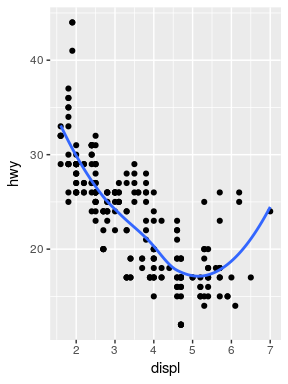
1. What geom would you use to draw a line chart? A boxplot? A histogram? An area chart?
2. Run this code in your head and predict what the output will look like. Then, run the code in R and check your predictions.

* ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +   
   geom\_point() +   
   geom\_smooth(se = FALSE)

1. What does show.legend = FALSE do? What happens if you remove it?  
   Why do you think I used it earlier in the chapter?
2. What does the se argument to geom\_smooth() do?
3. Will these two graphs look different? Why/why not?

* ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +   
   geom\_point() +   
   geom\_smooth()  
    
  ggplot() +   
   geom\_point(data = mpg, mapping = aes(x = displ, y = hwy)) +   
   geom\_smooth(data = mpg, mapping = aes(x = displ, y = hwy))

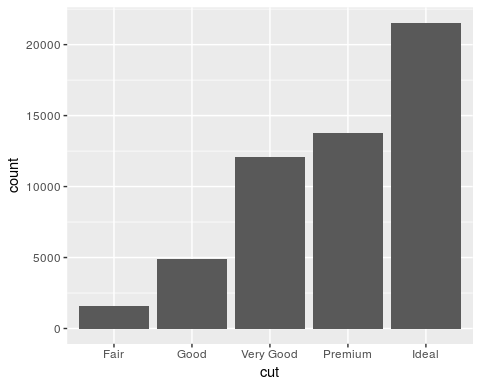
1. Recreate the R code necessary to generate the following graphs.

* 

## Statistical transformations

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with geom\_bar(). The following chart displays the total number of diamonds in the diamonds dataset, grouped by cut. The diamonds dataset comes in ggplot2 and contains information about ~54,000 diamonds, including the price, carat, color, clarity, and cut of each diamond. The chart shows that more diamonds are available with high quality cuts than with low quality cuts.

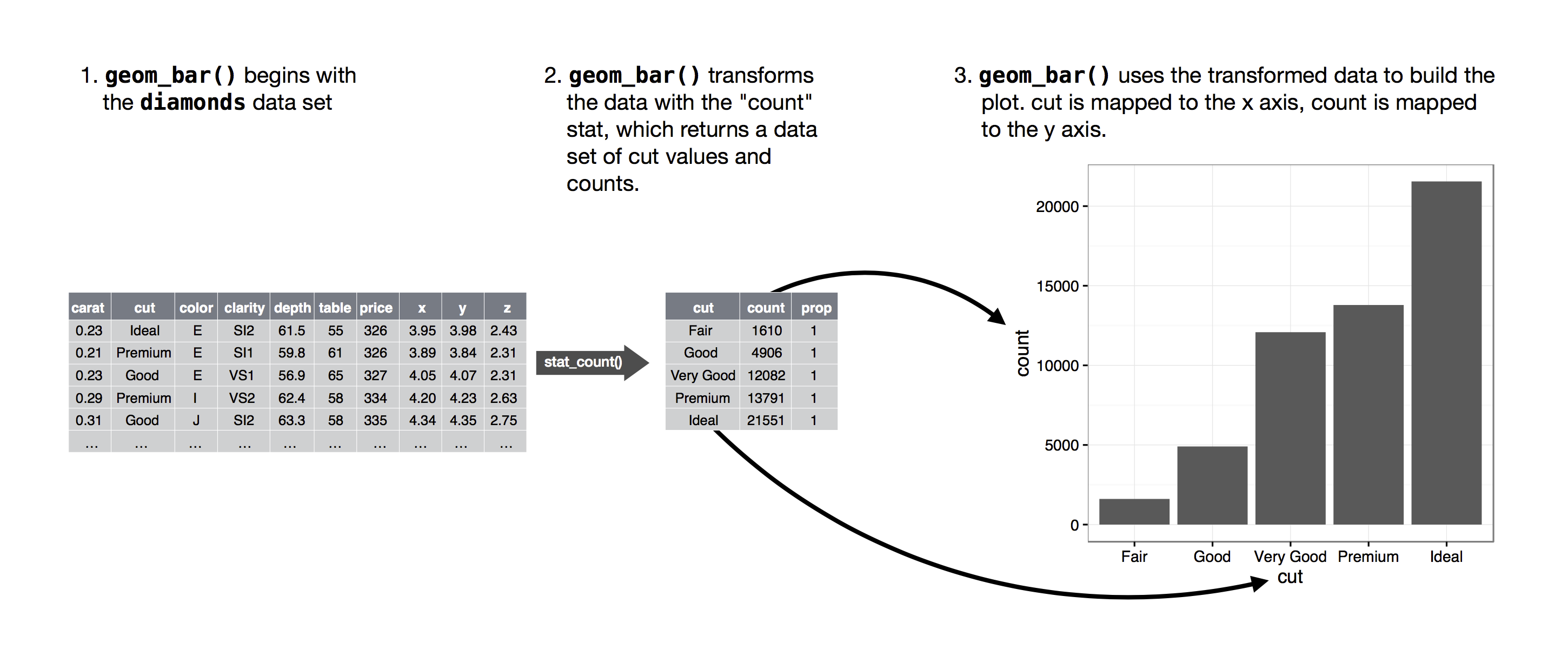
ggplot(data = diamonds) +   
 geom\_bar(mapping = aes(x = cut))



On the x-axis, the chart displays cut, a variable from diamonds. On the y-axis, it displays count, but count is not a variable in diamonds! Where does count come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

* bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
* smoothers fit a model to your data and then plot predictions from the model.
* boxplots compute a robust summary of the distribution and then display a specially formatted box.

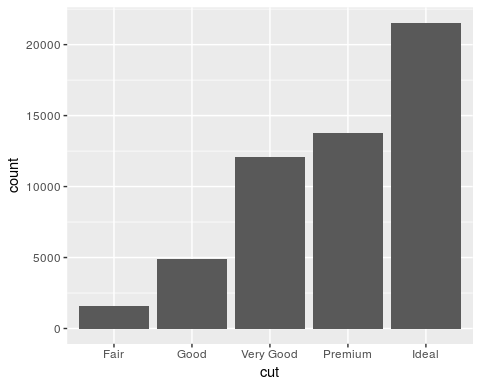
The algorithm used to calculate new values for a graph is called a **stat**, short for statistical transformation. The figure below describes how this process works with geom\_bar().



You can learn which stat a geom uses by inspecting the default value for the stat argument. For example, ?geom\_bar shows the default value for stat is "count", which means that geom\_bar() uses stat\_count(). stat\_count() is documented on the same page as geom\_bar(), and if you scroll down you can find a section called "Computed variables". That tells that it computes two new variables: count and prop.

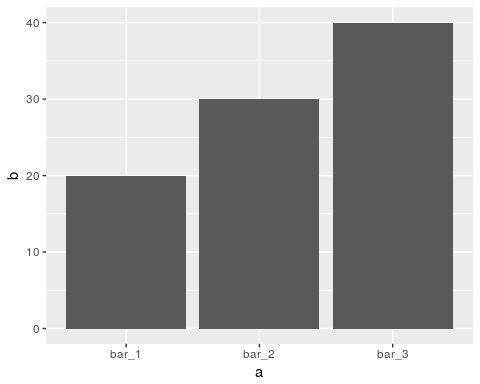
You can generally use geoms and stats interchangeably. For example, you can recreate the previous plot using stat\_count() instead of geom\_bar():

ggplot(data = diamonds) +   
 stat\_count(mapping = aes(x = cut))

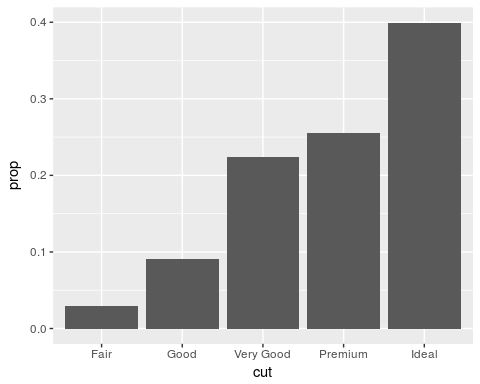


This works because every geom has a default stat; and every stat has a default geom. This means that you can typically use geoms without worrying about the underlying statistical transformation. There are three reasons you might need to use a stat explicitly:

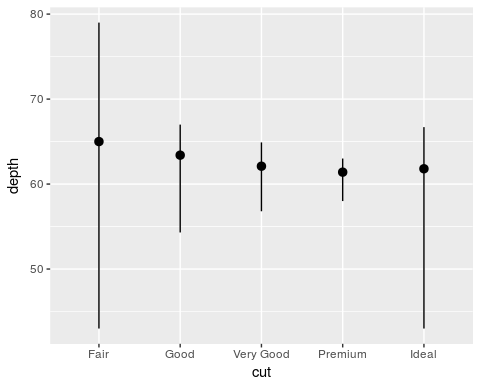
1. You might want to override the default stat. In the code below, I change the stat of geom\_bar() from count (the default) to identity. This lets me map the height of the bars to the raw values of a variable. Unfortunately when people talk about bar charts casually, they might be referring to this type of bar chart, where the height of the bar is already present in the data, or the previous bar chart where the height of the bar is generated by counting rows.

* demo <- tribble(  
   ~a, ~b,  
   "bar\_1", 20,  
   "bar\_2", 30,  
   "bar\_3", 40  
  )  
    
  ggplot(data = demo) +  
   geom\_bar(mapping = aes(x = a, y = b), stat = "identity")
* 
* (Don't worry that you haven't seen <- or tribble() before. You might be able to guess at their meaning from the context, and you'll learn exactly what they do soon!)

1. You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count:

* ggplot(data = diamonds) +   
   geom\_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
* 
* To find the variables computed by the stat, look for the help section titled "computed variables".

1. You might want to draw greater attention to the statistical transformation in your code. For example, you might use stat\_summary(), which summarises the y values for each unique x value, to draw attention to the summary that you're computing:

* ggplot(data = diamonds) +   
   stat\_summary(  
   mapping = aes(x = cut, y = depth),  
   fun.ymin = min,  
   fun.ymax = max,  
   fun.y = median  
   )
* 

ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in usual way, e.g. ?stat\_bin. To see a complete list of stats, try the ggplot2 cheatsheet.

### Exercises

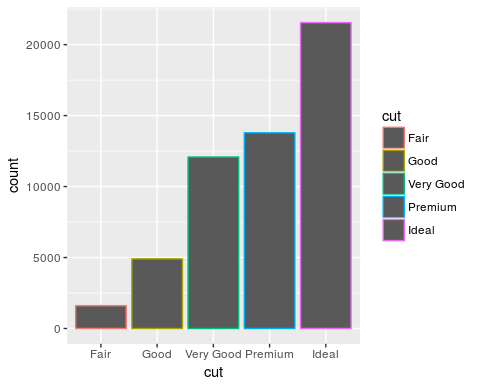
1. What is the default geom associated with stat\_summary()? How could you rewrite the previous plot to use that geom function instead of the stat function?
2. What does geom\_col() do? How is it different to geom\_bar()?
3. Most geoms and stats come in pairs that are almost always used in concert. Read through the documentation and make a list of all the pairs. What do they have in common?
4. What variables does stat\_smooth() compute? What parameters control its behaviour?
5. In our proportion bar chart, we need to set group = 1. Why? In other words what is the problem with these two graphs?

* ggplot(data = diamonds) +   
   geom\_bar(mapping = aes(x = cut, y = ..prop..))  
  ggplot(data = diamonds) +   
   geom\_bar(mapping = aes(x = cut, fill = color, y = ..prop..))

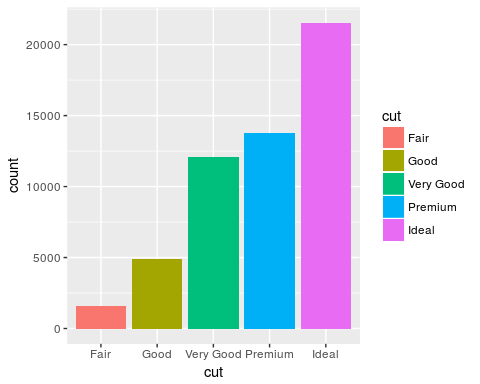
## Position adjustments

There's one more piece of magic associated with bar charts. You can colour a bar chart using either the colour aesthetic, or more usefully, fill:

ggplot(data = diamonds) +   
 geom\_bar(mapping = aes(x = cut, colour = cut))

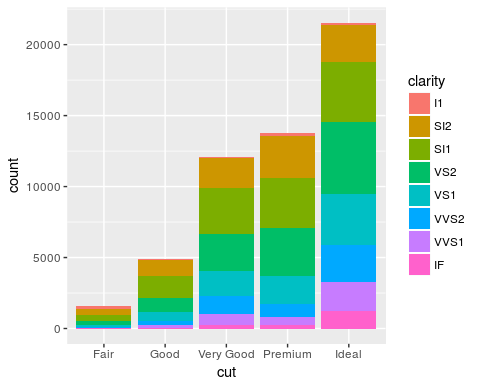


ggplot(data = diamonds) +   
 geom\_bar(mapping = aes(x = cut, fill = cut))

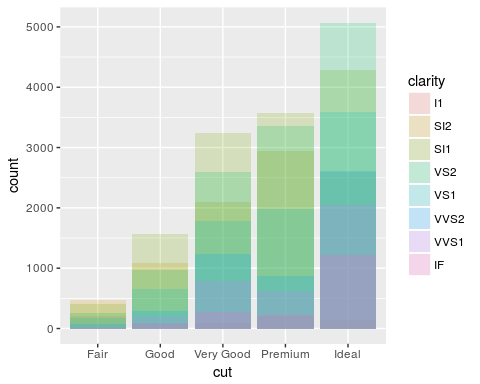
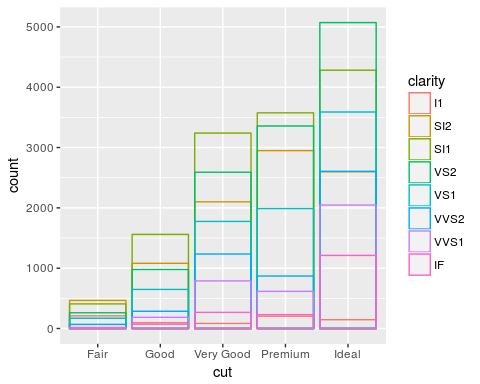
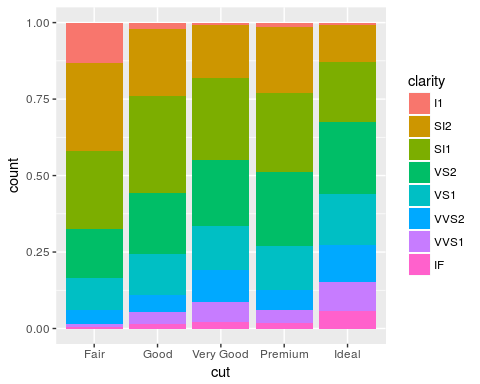
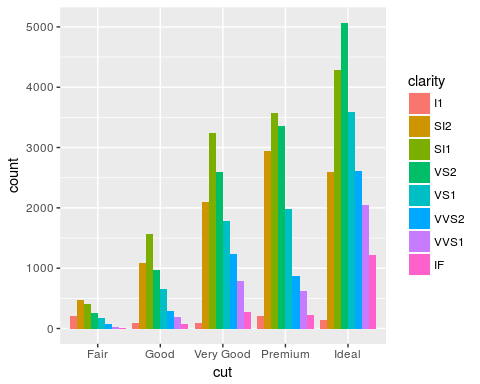


Note what happens if you map the fill aesthetic to another variable, like clarity: the bars are automatically stacked. Each colored rectangle represents a combination of cut and clarity.

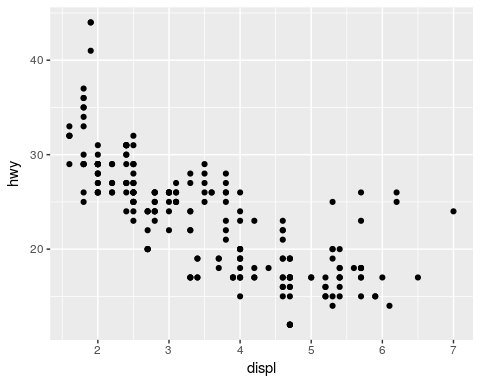
ggplot(data = diamonds) +   
 geom\_bar(mapping = aes(x = cut, fill = clarity))



The stacking is performed automatically by the **position adjustment** specified by the position argument. If you don't want a stacked bar chart, you can use one of three other options: "identity", "dodge" or "fill".

* position = "identity" will place each object exactly where it falls in the context of the graph. This is not very useful for bars, because it overlaps them. To see that overlapping we either need to make the bars slightly transparent by setting alpha to a small value, or completely transparent by setting fill = NA.
* ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) +   
   geom\_bar(alpha = 1/5, position = "identity")
* 
* ggplot(data = diamonds, mapping = aes(x = cut, colour = clarity)) +   
   geom\_bar(fill = NA, position = "identity")
* 
* The identity position adjustment is more useful for 2d geoms, like points, where it is the default.
* position = "fill" works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.
* ggplot(data = diamonds) +   
   geom\_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
* 
* position = "dodge" places overlapping objects directly *beside* one another. This makes it easier to compare individual values.
* ggplot(data = diamonds) +   
   geom\_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
* 

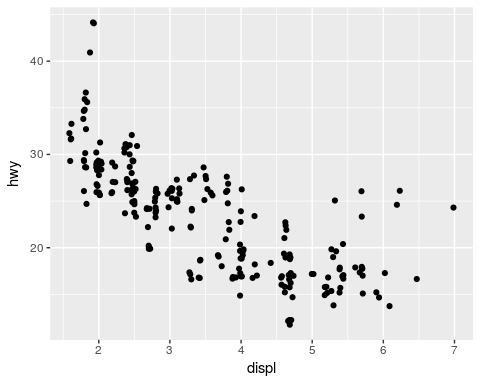
There's one other type of adjustment that's not useful for bar charts, but it can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?



The values of hwy and displ are rounded so the points appear on a grid and many points overlap each other. This problem is known as **overplotting**. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination of hwy and displ that contains 109 values?

You can avoid this gridding by setting the position adjustment to "jitter". position = "jitter" adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise.

ggplot(data = mpg) +   
 geom\_point(mapping = aes(x = displ, y = hwy), position = "jitter")

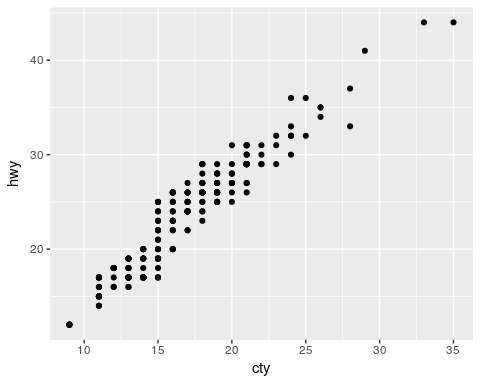


Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, ggplot2 comes with a shorthand for geom\_point(position = "jitter"): geom\_jitter().

To learn more about a position adjustment, look up the help page associated with each adjustment: ?position\_dodge, ?position\_fill, ?position\_identity, ?position\_jitter, and ?position\_stack.

### Exercises

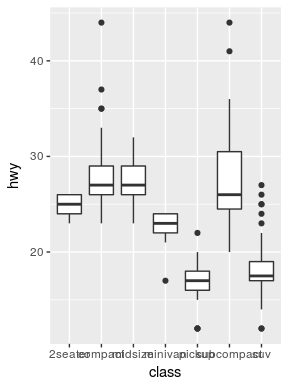
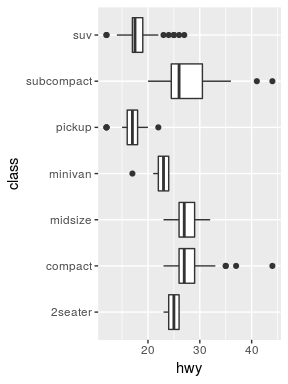
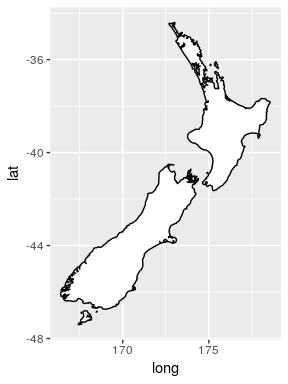
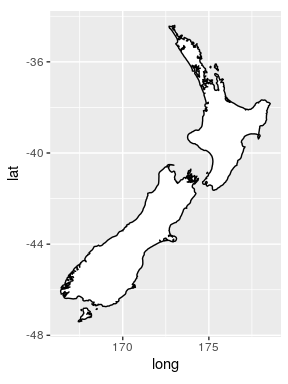
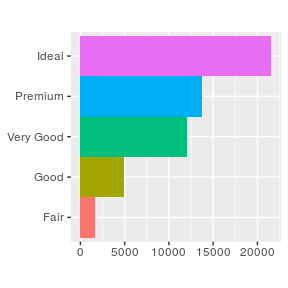
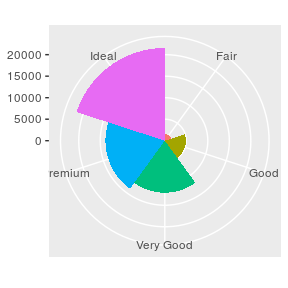
1. What is the problem with this plot? How could you improve it?

* ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +   
   geom\_point()
* 

1. What parameters to geom\_jitter() control the amount of jittering?
2. Compare and contrast geom\_jitter() with geom\_count().
3. What's the default position adjustment for geom\_boxplot()? Create a visualisation of the mpg dataset that demonstrates it.

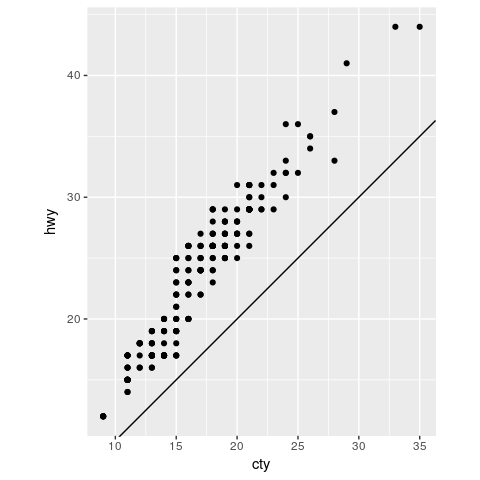
## Coordinate systems

Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y position act independently to find the location of each point. There are a number of other coordinate systems that are occasionally helpful.

* coord\_flip() switches the x and y axes. This is useful (for example), if you want horizontal boxplots. It's also useful for long labels: it's hard to get them to fit without overlapping on the x-axis.
* ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +   
   geom\_boxplot()
* 
* ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +   
   geom\_boxplot() +  
   coord\_flip()
* 
* coord\_quickmap() sets the aspect ratio correctly for maps. This is very important if you're plotting spatial data with ggplot2 (which unfortunately we don't have the space to cover in this book).
* nz <- map\_data("nz")  
    
  ggplot(nz, aes(long, lat, group = group)) +  
   geom\_polygon(fill = "white", colour = "black")
* 
* ggplot(nz, aes(long, lat, group = group)) +  
   geom\_polygon(fill = "white", colour = "black") +  
   coord\_quickmap()
* 
* coord\_polar() uses polar coordinates. Polar coordinates reveal an interesting connection between a bar chart and a Coxcomb chart.
* bar <- ggplot(data = diamonds) +   
   geom\_bar(  
   mapping = aes(x = cut, fill = cut),   
   show.legend = FALSE,  
   width = 1  
   ) +   
   theme(aspect.ratio = 1) +  
   labs(x = NULL, y = NULL)  
    
  bar + coord\_flip()
* 
* bar + coord\_polar()
* 

### Exercises

1. Turn a stacked bar chart into a pie chart using coord\_polar().
2. What does labs() do? Read the documentation.
3. What's the difference between coord\_quickmap() and coord\_map()?
4. What does the plot below tell you about the relationship between city and highway mpg? Why is coord\_fixed() important? What does geom\_abline() do?

* ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +  
   geom\_point() +   
   geom\_abline() +  
   coord\_fixed()
* 

## The layered grammar of graphics

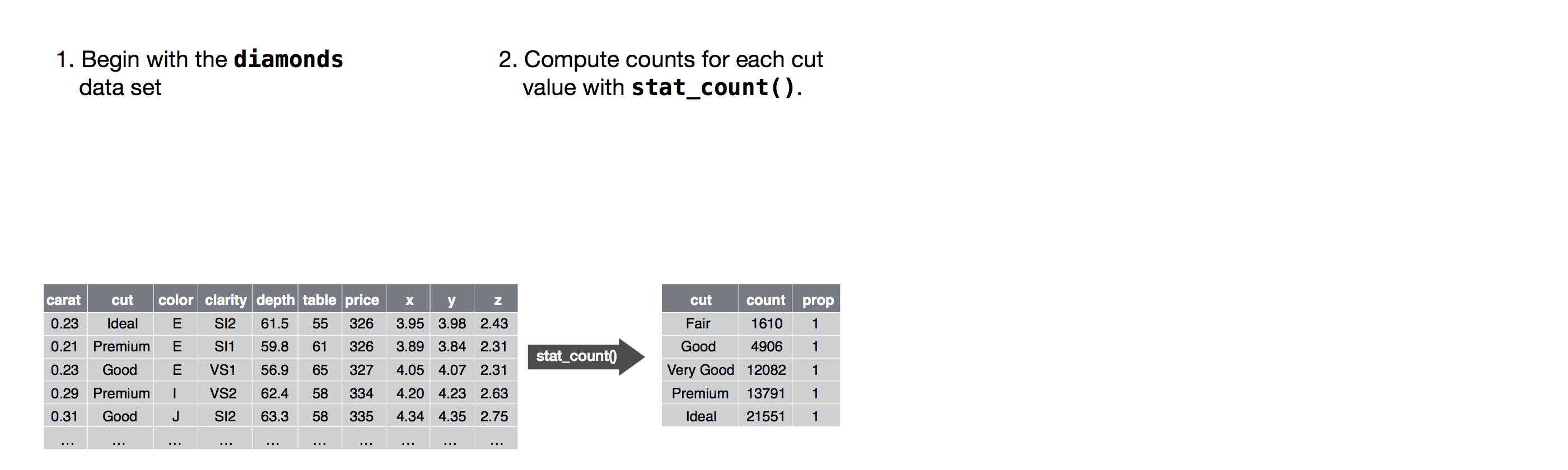
In the previous sections, you learned much more than how to make scatterplots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with ggplot2. To see this, let's add position adjustments, stats, coordinate systems, and faceting to our code template:

ggplot(data = <DATA>) +   
 <GEOM\_FUNCTION>(  
 mapping = aes(<MAPPINGS>),  
 stat = <STAT>,   
 position = <POSITION>  
 ) +  
 <COORDINATE\_FUNCTION> +  
 <FACET\_FUNCTION>

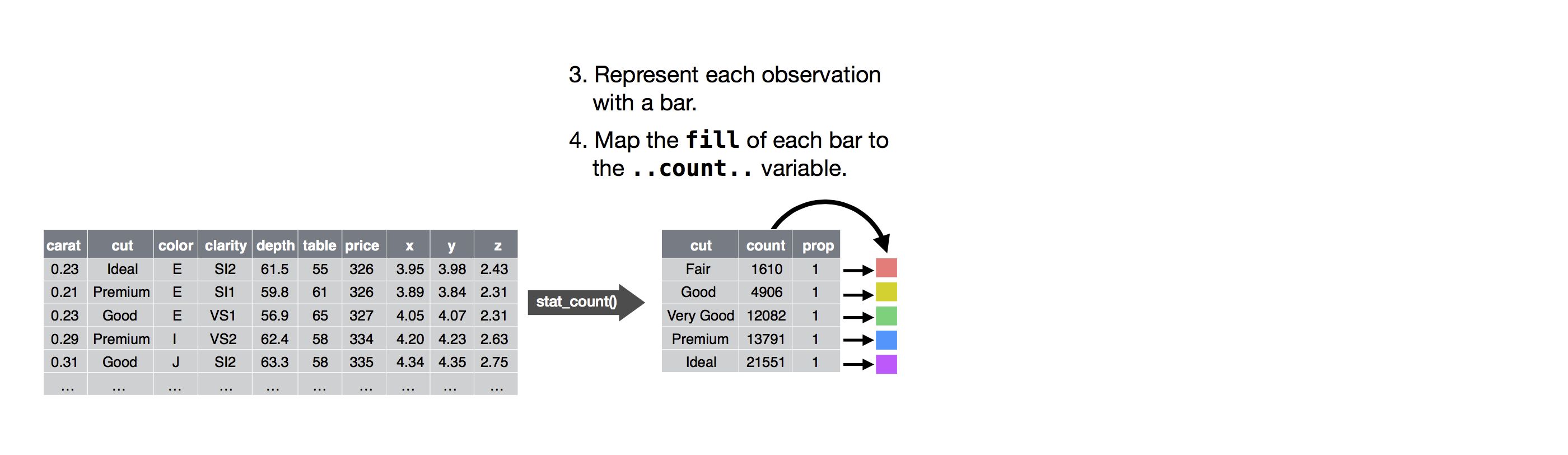
Our new template takes seven parameters, the bracketed words that appear in the template. In practice, you rarely need to supply all seven parameters to make a graph because ggplot2 will provide useful defaults for everything except the data, the mappings, and the geom function.

The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system, and a faceting scheme.

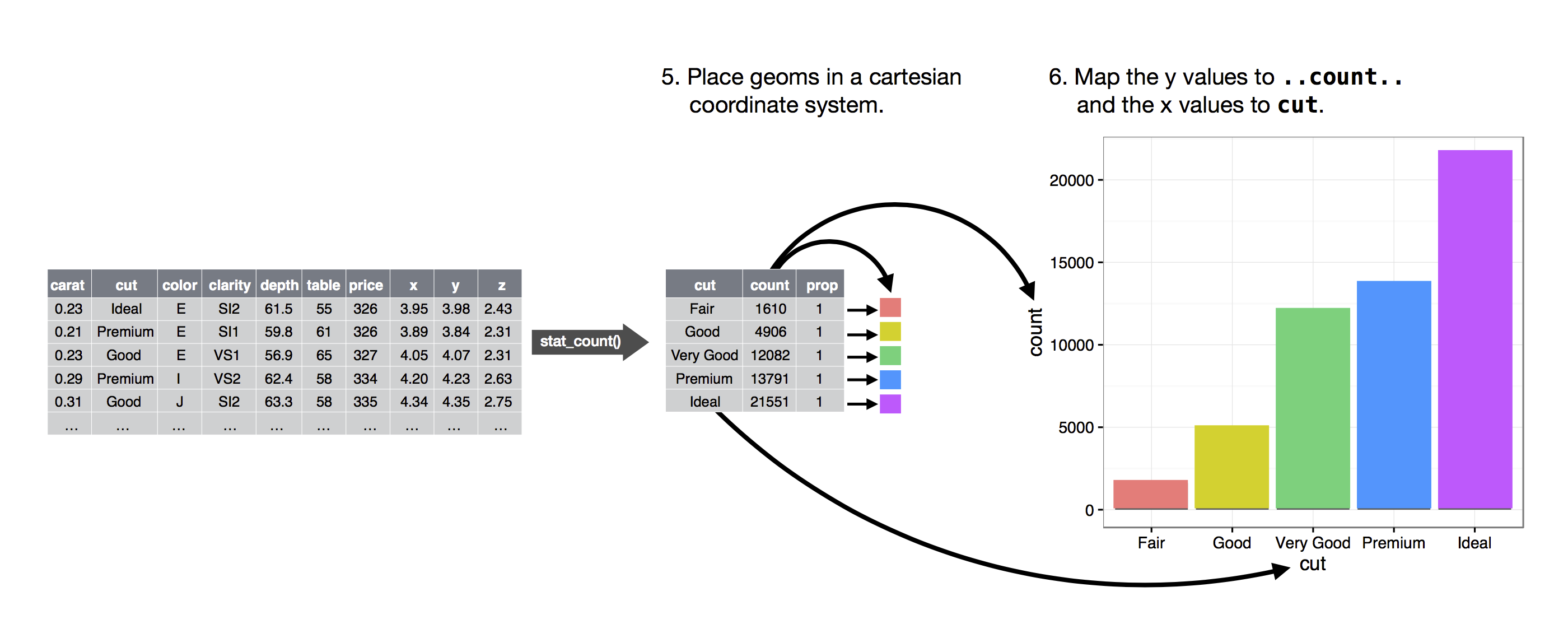
To see how this works, consider how you could build a basic plot from scratch: you could start with a dataset and then transform it into the information that you want to display (with a stat).



Next, you could choose a geometric object to represent each observation in the transformed data. You could then use the aesthetic properties of the geoms to represent variables in the data. You would map the values of each variable to the levels of an aesthetic.



You'd then select a coordinate system to place the geoms into. You'd use the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables. At that point, you would have a complete graph, but you could further adjust the positions of the geoms within the coordinate system (a position adjustment) or split the graph into subplots (faceting). You could also extend the plot by adding one or more additional layers, where each additional layer uses a dataset, a geom, a set of mappings, a stat, and a position adjustment.



You could use this method to build *any* plot that you imagine. In other words, you can use the code template that you've learned in this chapter to build hundreds of thousands of unique plots.

## Learn More

* [ggplot2 tutorial](https://github.com/jennybc/ggplot2-tutorial) by Prof Jenny Bryan
* [R4DS Graphics for communication](http://r4ds.had.co.nz/graphics-for-communication.html)
* [The ggplot2 Cheat Sheet - RStudio](https://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf)
* [ggplot2: Elegant Graphics for Data Analysis](https://amzn.com/331924275X)
* [R Graphics Cookbook](http://www.cookbook-r.com/Graphs/)

# Model with Split-Apply-Combine Strategy

## Readings

1. [Wickham, Hadley, 2011. The Split-Apply-Combine Strategy for Data Analysis, Journal of Statistical Software, Volume 40, Issue 1.](https://www.jstatsoft.org/article/view/v040i01/v40i01.pdf)
2. [R4DS Many Models](http://r4ds.had.co.nz/many-models.html)

If you are not familiar with the basic workflow of fitting models with R, you should review two chapters:

1. [R4DS Model basics](http://r4ds.had.co.nz/model-basics.html)
2. [R4DS Model building](http://r4ds.had.co.nz/model-building.html)

## Think before you create excerpts of your data ...

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

snippet <- my\_big\_dataset %>%  
 filter(some\_variable == some\_value)  
## or  
snippet <- subset(my\_big\_dataset, some\_variable == some\_value)

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 facetting in ggplot2 -- **don’t subset the data**. Or, more realistically, 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 one of these strategies that subset at "compute time" will get the job done:

* Pass an intact data frame and use the subset = argument of a command. Many functions have it!
* ## regress life expectancy on year for Canada  
  canada\_fit <-  
   lm(lifeExp ~ year, data = gapminder, subset = country == "Canada")
* Pipe filtered data into a command. A very general solution.
* ## compare gdpPercap in Australia vs New Zealand  
  oceania\_ttest <- gapminder %>%  
   filter(continent == "Asia") %>%   
   t.test(gdpPercap ~ country, data = .)

Creating copies and excerpts of your data clutters up your workspace, script, and mind, often leading to confusion and mistakes. It can be useful during development, but try to eliminate this as you polish your analysis.

### Load dplyr, tidyr, purrr and gapminder

I choose to load the entire tidyverse. Also load gapminder.

library(gapminder)  
library(tidyverse)

### Split Apply Combine

A common analytical pattern is to

* split data into pieces,
* apply some function to each piece,
* combine the results back together again.

R has specialized tools for this that are much nicer than whatever DIY approach you might be contemplating. Especially if it involves for() loops. There's nothing inherently wrong or evil about for() loops, but you will spend alot of time and characters on bookkeeping tasks. That can be a great way to pilot a computation, because it is so blessedly concrete. But consider revisiting your implementation with higher-level data aggregation tools once you've established proof-of-principle.

In base R, these are the "apply" functions, such as apply(), aggregate(), tapply(), and by(). We prefer to use similar tools from the tidyverse, due to a more consistent interface, analysis-friendly output, and a more concise way to describe what to compute. This article by Hadley Wickham -- [The split-apply-combine strategy for data analysis](http://www.jstatsoft.org/v40/i01/paper) -- provides a good high-level overview, but know that the tidyverse approaches outlined here now supercede the plyr package described there.

### Entry level: dplyr::group\_by()

The most lightweight solution is given by dplyr::group\_by(). group\_by() adds some extra grouping structure to a data frame, based on levels of one or more categorical variables, but leaves the data frame intact. Then you can do group-wise computations using various functions in the tidyverse that automatically honor these groups.

The main function here is summarize(), which collapses each group into a single row in a new group-summarized data frame.

When does this break down? If anything you want to compute for a group is more complicated than, say, a single number, you have outgrown dplyr::summarize(). Example: getting the range of life expectancies seen for each continent doesn't work with the code below.

gapminder %>%  
 group\_by(continent) %>%  
 summarize(range = range(lifeExp))

Now, in this case, we could reframe this in a summarize()-friendly form, but that doesn't work in general.

gapminder %>%  
 group\_by(continent) %>%  
 summarize\_each(funs(min, max), lifeExp)

## # A tibble: 5 × 3  
## continent min max  
## <fctr> <dbl> <dbl>  
## 1 Africa 23.599 76.442  
## 2 Americas 37.579 80.653  
## 3 Asia 28.801 82.603  
## 4 Europe 43.585 81.757  
## 5 Oceania 69.120 81.235

### General approach: nesting

We use **nesting** as an extension of **grouping** in order to support more general group-wise computation. The collapse to a single row per group happens right away here, unlike the simple grouping above. When you nest data, the non-grouping variables are packaged into group-specific data frames that are held in a special variable called a list-column. You then apply your computation to the components of this list, i.e. the data frames. List-columns take a little getting used to, but the payoff is huge.

Let's get ready to fit a model for each country in the Gapminder dataset. First we group, as above, and then nest. We group by country and continent in order to keep those two variables "on the outside".

(gap\_nested <- gapminder %>%   
 group\_by(continent, country) %>%   
 nest())

## # A tibble: 142 × 3  
## continent country data  
## <fctr> <fctr> <list>  
## 1 Asia Afghanistan <tibble [12 × 4]>  
## 2 Europe Albania <tibble [12 × 4]>  
## 3 Africa Algeria <tibble [12 × 4]>  
## 4 Africa Angola <tibble [12 × 4]>  
## 5 Americas Argentina <tibble [12 × 4]>  
## 6 Oceania Australia <tibble [12 × 4]>  
## 7 Europe Austria <tibble [12 × 4]>  
## 8 Asia Bahrain <tibble [12 × 4]>  
## 9 Asia Bangladesh <tibble [12 × 4]>  
## 10 Europe Belgium <tibble [12 × 4]>  
## # ... with 132 more rows

What do you notice?

* The immediate collapse to 142 rows, one per country.
* The familiar presence of continent and country.
* The unfamiliar look of the new data variable, which is ... a list! It is a list-column of continent-specific tibbles.

How on earth do you inspect it?!? In RStudio, gap\_nested %>% View() is often helpful (moderately so in this case). It's often nicer to inspect a single element like so:

gap\_nested[[1, "data"]]

## # A tibble: 12 × 4  
## year lifeExp pop gdpPercap  
## <int> <dbl> <int> <dbl>  
## 1 1952 28.801 8425333 779.4453  
## 2 1957 30.332 9240934 820.8530  
## 3 1962 31.997 10267083 853.1007  
## 4 1967 34.020 11537966 836.1971  
## 5 1972 36.088 13079460 739.9811  
## 6 1977 38.438 14880372 786.1134  
## 7 1982 39.854 12881816 978.0114  
## 8 1987 40.822 13867957 852.3959  
## 9 1992 41.674 16317921 649.3414  
## 10 1997 41.763 22227415 635.3414  
## 11 2002 42.129 25268405 726.7341  
## 12 2007 43.828 31889923 974.5803

[Remember that](block004_basic-r-objects.html) double square brackets can only be used to extract a single element and they are simplifying. An equivalent call is gap\_nested[["data"]][[1]] (which I find even more opaque) or gap\_nested$data[[1]] (which is pretty nice). We're looking at the first of 142 country-specific data frames, which happens to be for Afghanistan.

The presence of list-columns is always a temporary, uncomfortable state.

### Apply a function purrr::map() and mutate()

How do we now iterate over the elements of gap\_nested$data? It is a list, so we use general approaches for applying a function to each element of a list. In base R, this means lapply(), but we will use the tidyverse function purrr::map(), which has a few advantages [described elsewhere](https://speakerdeck.com/jennybc/purrr-tutorial-intro-very-rough-draft).

Walk before you run, i.e. do an example first!

Let's fit a model to the data from Afghanistan. The form of the right-hand side is so that our intercept has a nice interpretation.

(fit <- lm(lifeExp ~ I(year - 1950), data = gap\_nested[[1, "data"]]))

##   
## Call:  
## lm(formula = lifeExp ~ I(year - 1950), data = gap\_nested[[1,   
## "data"]])  
##   
## Coefficients:  
## (Intercept) I(year - 1950)   
## 29.3566 0.2753

Let's capture that logic in a function and informally test that it returns the same results for Afghanistan.

le\_vs\_yr <- function(df) {  
 lm(lifeExp ~ I(year - 1950), data = df)  
}  
le\_vs\_yr(gap\_nested[[1, "data"]])

##   
## Call:  
## lm(formula = lifeExp ~ I(year - 1950), data = df)  
##   
## Coefficients:  
## (Intercept) I(year - 1950)   
## 29.3566 0.2753

After you walk, jog before you run. Use purrr::map() to apply the fitting function le\_vs\_yr() to the first two elements of gap\_nested$data.

fits <- purrr::map(gap\_nested$data[1:2], le\_vs\_yr)  
fits

## [[1]]  
##   
## Call:  
## lm(formula = lifeExp ~ I(year - 1950), data = df)  
##   
## Coefficients:  
## (Intercept) I(year - 1950)   
## 29.3566 0.2753   
##   
##   
## [[2]]  
##   
## Call:  
## lm(formula = lifeExp ~ I(year - 1950), data = df)  
##   
## Coefficients:  
## (Intercept) I(year - 1950)   
## 58.5598 0.3347

So, how do we run, i.e. scale this up to all countries? And where do we put these fitted models? We'll use purrr::map() inside mutate(), meaning we store the models inside gap\_nested in another list-column.

(gap\_nested <- gap\_nested %>%   
 mutate(fit = purrr::map(data, le\_vs\_yr)))

## # A tibble: 142 × 4  
## continent country data fit  
## <fctr> <fctr> <list> <list>  
## 1 Asia Afghanistan <tibble [12 × 4]> <S3: lm>  
## 2 Europe Albania <tibble [12 × 4]> <S3: lm>  
## 3 Africa Algeria <tibble [12 × 4]> <S3: lm>  
## 4 Africa Angola <tibble [12 × 4]> <S3: lm>  
## 5 Americas Argentina <tibble [12 × 4]> <S3: lm>  
## 6 Oceania Australia <tibble [12 × 4]> <S3: lm>  
## 7 Europe Austria <tibble [12 × 4]> <S3: lm>  
## 8 Asia Bahrain <tibble [12 × 4]> <S3: lm>  
## 9 Asia Bangladesh <tibble [12 × 4]> <S3: lm>  
## 10 Europe Belgium <tibble [12 × 4]> <S3: lm>  
## # ... with 132 more rows

We have a new list-column fit that is even more intimidating than the previous, data. The fit variable holds 142 fitted linear models. What shall we do with that?

### Always have an exit strategy

The list-column state is an uncomfortable and temporary one. The goal is always to pull information out of these complicated objects and package as something simpler, usually a tibble. You always want a list-column exit strategy!

We are not afraid to fit linear models in this example because there is a fantastic package called [broom](https://github.com/dgrtwo/broom) from David Robinson that provides exactly this for lm() and many other functions.

broom provides three key functions that take a lm() fit as input and give a useful tibble back:

* tidy(): a tidy version of summary(), e.g. a table with one row per parameter estimate
* augment(): the original data, augmented with columns such as fitted values and residuals
* glance(): a one-row model summary

All of these are much friendlier than a fitted lm() object and set us up for interesting downstream analyses and plots.

### Simplify and combine

Let's look at the result of broom::tidy() for a single model. Walk before you run.

library(broom)  
tidy(gap\_nested$fit[[1]])

## term estimate std.error statistic p.value  
## 1 (Intercept) 29.3566375 0.69898128 41.99918 1.404235e-12  
## 2 I(year - 1950) 0.2753287 0.02045093 13.46289 9.835213e-08

We get a two row data frame, one with results for the intercept and one for the slope. This is much more approachable than fitted lm objects!

Apply tidy() to the model for each country with the same purrr::map() inside mutate() strategy as above.

(gap\_nested <- gap\_nested %>%   
 mutate(tidy = purrr::map(fit, tidy)))

## # A tibble: 142 × 5  
## continent country data fit tidy  
## <fctr> <fctr> <list> <list> <list>  
## 1 Asia Afghanistan <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 2 Europe Albania <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 3 Africa Algeria <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 4 Africa Angola <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 5 Americas Argentina <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 6 Oceania Australia <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 7 Europe Austria <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 8 Asia Bahrain <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 9 Asia Bangladesh <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## 10 Europe Belgium <tibble [12 × 4]> <S3: lm> <data.frame [2 × 5]>  
## # ... with 132 more rows

The last step is now to simplify, preferably back to a normal tibble. We do this by retaining variables that are amenable to simplification and using unnest(), thus completing the circle.

(gap\_coefs <- gap\_nested %>%   
 select(continent, country, tidy) %>%   
 unnest(tidy))

## # A tibble: 284 × 7  
## continent country term estimate std.error statistic  
## <fctr> <fctr> <chr> <dbl> <dbl> <dbl>  
## 1 Asia Afghanistan (Intercept) 29.3566375 0.698981278 41.999176  
## 2 Asia Afghanistan I(year - 1950) 0.2753287 0.020450934 13.462890  
## 3 Europe Albania (Intercept) 58.5597618 1.133575812 51.659325  
## 4 Europe Albania I(year - 1950) 0.3346832 0.033166387 10.091036  
## 5 Africa Algeria (Intercept) 42.2364149 0.756269040 55.848399  
## 6 Africa Algeria I(year - 1950) 0.5692797 0.022127070 25.727749  
## 7 Africa Angola (Intercept) 31.7079741 0.804287463 39.423683  
## 8 Africa Angola I(year - 1950) 0.2093399 0.023532003 8.895964  
## 9 Americas Argentina (Intercept) 62.2250191 0.167091314 372.401279  
## 10 Americas Argentina I(year - 1950) 0.2317084 0.004888791 47.395847  
## # ... with 274 more rows, and 1 more variables: p.value <dbl>

### Recap

The whole point of this was to get apply a computation to all the pieces of a dataset and glue the results back together. First, let's review all of our work so far in one place. It's remarkably compact.

gap\_nested <- gapminder %>%   
 group\_by(continent, country) %>%   
 nest()  
  
le\_vs\_yr <- function(df) {  
 lm(lifeExp ~ I(year - 1950), data = df)  
}  
  
gap\_coefs <- gap\_nested %>%   
 mutate(fit = purrr::map(data, le\_vs\_yr),  
 tidy = purrr::map(fit, tidy)) %>%   
 select(continent, country, tidy) %>%   
 unnest(tidy)

Reflect on how you would have obtained a data frame of country-specific intercepts and slopes from a regression of life expectancy on year. Did you know any approaches for solving this problem? If no, then rejoice that you now have one! If yes, does the approach outlined here seem simpler?

List-columns take some getting used to, but they are a required component of the strategy laid out above for general-purpose split-apply-combine.

### Enjoy the payoff of your work

Let's celebrate by exploring the estimated slopes and intercepts a bit. First we recode the variable corresponding to "intercept" vs "slope".

(gap\_coefs <- gap\_coefs %>%  
 mutate(term = recode(term,  
 `(Intercept)` = "intercept",  
 `I(year - 1950)` = "slope")))

## # A tibble: 284 × 7  
## continent country term estimate std.error statistic  
## <fctr> <fctr> <chr> <dbl> <dbl> <dbl>  
## 1 Asia Afghanistan intercept 29.3566375 0.698981278 41.999176  
## 2 Asia Afghanistan slope 0.2753287 0.020450934 13.462890  
## 3 Europe Albania intercept 58.5597618 1.133575812 51.659325  
## 4 Europe Albania slope 0.3346832 0.033166387 10.091036  
## 5 Africa Algeria intercept 42.2364149 0.756269040 55.848399  
## 6 Africa Algeria slope 0.5692797 0.022127070 25.727749  
## 7 Africa Angola intercept 31.7079741 0.804287463 39.423683  
## 8 Africa Angola slope 0.2093399 0.023532003 8.895964  
## 9 Americas Argentina intercept 62.2250191 0.167091314 372.401279  
## 10 Americas Argentina slope 0.2317084 0.004888791 47.395847  
## # ... with 274 more rows, and 1 more variables: p.value <dbl>

Due to the way we parametrized the model, the intercepts correspond to expected life expectancy in 1950. These numbers should be plausible as human life expectancies. The slopes correspond to change in expected life expectancy from one year to the next. We expect positive numbers to dominate, but they'll probably less than one. A reshaped version of the estimates, gap\_ests, is handy for numerical summarization and visualization.

(gap\_ests <- gap\_coefs %>%   
 select(continent:estimate) %>%   
 spread(key = term, value = estimate))

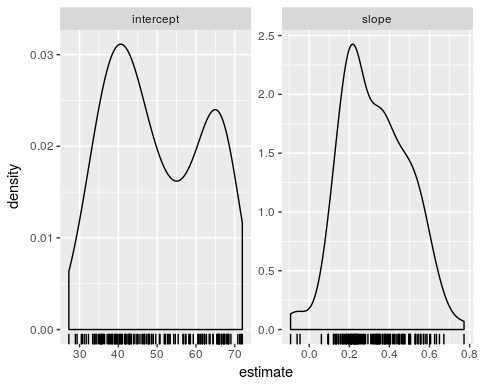
## # A tibble: 142 × 4  
## continent country intercept slope  
## \* <fctr> <fctr> <dbl> <dbl>  
## 1 Africa Algeria 42.23641 0.56927972  
## 2 Africa Angola 31.70797 0.20933986  
## 3 Africa Benin 38.92005 0.33423287  
## 4 Africa Botswana 52.80778 0.06066853  
## 5 Africa Burkina Faso 33.95674 0.36397483  
## 6 Africa Burundi 40.27037 0.15413427  
## 7 Africa Cameroon 40.74917 0.25014685  
## 8 Africa Central African Republic 38.44170 0.18390559  
## 9 Africa Chad 39.30288 0.25324406  
## 10 Africa Comoros 39.09522 0.45039091  
## # ... with 132 more rows

gap\_ests %>%   
 select(intercept, slope) %>%   
 summary()

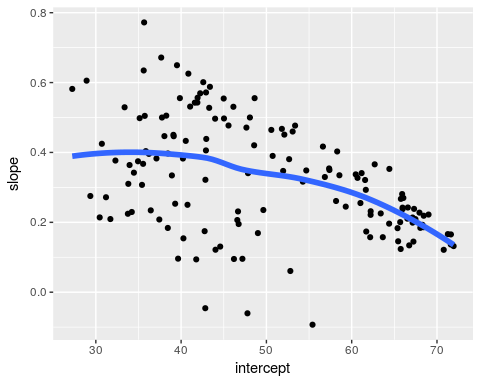
## intercept slope   
## Min. :27.24 Min. :-0.09302   
## 1st Qu.:39.36 1st Qu.: 0.20832   
## Median :47.42 Median : 0.32145   
## Mean :49.86 Mean : 0.32590   
## 3rd Qu.:62.05 3rd Qu.: 0.44948   
## Max. :71.95 Max. : 0.77218

The numerical summaries look reasonable. We conclude with a look at the full distribution.

ggplot(gap\_coefs, aes(x = estimate)) +  
 geom\_density() + geom\_rug() + facet\_wrap(~ term, scales = "free")



ggplot(gap\_ests, aes(x = intercept, y = slope)) +  
 geom\_point() +  
 geom\_smooth(se = FALSE, lwd = 2)



# RMarkdown

## Readings

1. [W] Chapter 30, 29

knitr is an R package that allows you to organize your notes, code, and results in a single document. It's a great tool for "literate programming" -- the idea that your code should be readable by humans as well as computers! It also keeps your writing and results together, so if you collect some new data or change how you clean the data, you just have to re-compile the document and you're up to date!

You write knitr documents in a simple plain text-like format called markdown, which allows you to format text using intuitive notation, so that you can focus on the content you're writing. But you still get a well-formatted document out. In fact, you can turn your plain text (and R code and results) into an html file or, if you have an installation of LaTeX and Pandoc on your machine, a pdf, or even a Word document (if you must!).

To get started, install the knitr package.

install.packages("knitr")

When you click on File -> New File, there is an option for "R Markdown...". Choose this and accept the default options in the dialog box that follows (but note that you can also create presentations this way). Save the file and click on the "Knit HTML" button at the top of the script. Compare the output to the source.

**Formatting Text in Markdown**

Visit <http://rmarkdown.rstudio.com/authoring_basics.html> and briefly check out some of the formatting options.

In the example document add

* Headers using #
* Emphasis using astericks: \*italics\* and \*\*bold\*\*
* Lists using \* and numbered lists using 1., 2., etc.
* **Bonus:** Create a table {: .challenge}

Markdown also supports LaTeX equation editing. We can display pretty equations by enclosing them in $, e.g., $\alpha = \dfrac{1}{(1 - \beta)^2}$ renders as: .

The top of the source (.Rmd) file has some header material in YAML format (enclosed by triple dashes). Some of this gets displayed in the output header, other of it provides formatting information to the conversion engine.

To distinguish R code from text, RMarkdown uses three back-ticks followed by {r} to distinguish a "code chunk". In RStudio, the keyboard shortcut to create a code chunk is command-option-i or control-alt-i.

A code chunk will set off the code and its results in the output document, but you can also print the results of code within a text block by enclosing code like so: `r code-here`.

**Use knitr to Produce a Report**

1. Open an new .Rmd script and save it as inflammation\_report.Rmd
2. Copy code from earlier into code chunks to read the inflammation data and plot average inflammation.
3. Add a few notes describing what the code does and what the main findings are. Include an in-line calculation of the median inflammation level.
4. knit the document and view the html result. {: .challenge}

## Overview

This describes a hands-on activity where the goal is to author an R Markdown document and render it to HTML/pdf/docx.

For people also using Git and GitHub, we discuss how to keep the intermediate Markdown file, the figures, and what to commit to Git and push to GitHub.

Here is the official R Markdown documentation: <http://rmarkdown.rstudio.com>

## Step 0: Software installation and configuration

We assume the following

* You've already [installed Git](git01_git-install.html) and (probably) [a Git client](git02_git-clients.html).
* You've already [registered a free GitHub account](cm001_course-intro-sw-install-account-signup.html#register-a-github-account).
* You've already [installed R and RStudio](block000_r-rstudio-install.html).
* You've [tested your installation and configuration of Git, GitHub, and RStudio](git07_git-github-rstudio.html)
* You've completed [Homework 01](hw01_edit-README.html) or an equivalent exercise to truly test all of the above and to introduce you to Markdown.

## Step 1: Get ready to work

Launch RStudio, probably in the Project that corresponds to the repository where you are keeping all STAT 545 coursework. Make sure the workspace is clean and you've launched a fresh R process. Make sure the working directory is sensible.

Git people: we assume you are working inside a Git repo -- probably one that is already set up to push/pull with a GitHub remote. That is [covered elsewhere](git07_git-github-rstudio.html).

## Step 2: Practice with RStudio's boilerplate R Markdown document

I am modelling "walk before you run" here. It is best, especially for novices, to increase complexity in small increments. We will test our system's ability to render the ["hello world"](http://en.wikipedia.org/wiki/%22Hello,_world!%22_program) of R Markdown documents before we muddy the waters with our own, probably buggy, documents.

Do this: *File > New File > R Markdown ...*

* Give it an informative title. This will appear in the document but does not necessarily have anything to do with the file's name. But the title and filename should be similar! The title is for human eyeballs, so it can contain spaces and punctuation. The filename is for humans and computers, so it should have similar words in it but no spaces and no punctuation.
* Accept the default Author or edit if you wish.
* Accept the default output format of HTML.
* Click OK.

Save this document to a reasonable filename and location. The filename should end in .Rmd or .rmd. I **highly recommend** saving in the top-level of the directory that is also also a Git repository for your coursework and that is also an RStudio project and that is also current working directory. Trust me on this.

Git people: this is a decent time to commit your new Rmd.

Click on "Knit HTML" or do *File > Knit Document*. RStudio should display a preview of the resulting HTML. Also look at the file browser (which should be pointed at the directory where you saved the .Rmd file). You should see the R Markdown document, i.e. foo.Rmd AND the resulting HTML foo.html.

Congratulations, you've just made your first reproducible report with R Markdown.

Git people: this is a decent time to commit your new HTML. One day you might reconsider whether you want HTML in your repo, but don't worry about that now.

## Step 3: Take control of the output format

Do you really want HTML? Do you only want HTML? If so, you can skip this step!

The magical process that turns your R Markdown to HTML is like so: foo.Rmd --> foo.md --> foo.html. Note the intermediate markdown, foo.md. By default RStudio discards this, but you might want to hold on to that markdown.

Why? GitHub gives very special treatment to markdown files. They are rendered in an almost HTML-like way. This is great because it preserves all the charms of plain text but gives you a pseudo-webpage for free when you visit the file in the browser. In contrast, HTML is rendered as plain text on GitHub and you'll have to take special measures to see it the way you want.

In many cases, you *only want the markdown*. In that case, we switch the output format to github\_document. This means render will be foo.Rmd --> foo.md, where foo.md is GitHub-flavored markdown. If you still want the HTML *but also the intermediate markdown*, there's a way to request that too.

**Output format** is one of the many things we can control in the YAML frontmatter -- the text at the top of your file between leading and trailing lines of ---.

You can make some changes via the RStudio IDE: click on the "gear" in the top bar of the source editor, near the "Knit HTML" button. Select "Output options" and go to the Advanced tab and check "Keep markdown source file." Your YAML should now look more like this:

---   
 title: "Something fascinating"   
 author: "Jenny Bryan"   
 date: "2017-03-15"  
 output:   
 html\_document:   
 keep\_md: true   
 ---

You should have gained the line keep\_md: true. You can also simply edit the file yourself to achieve this.

In fact this hand-edit is necessary if you want to keep only markdown and get GitHub-flavored markdown. In that case, make your YAML look like this:

---   
 title: "Something fascinating"   
 author: "Jenny Bryan"   
 date: "2017-03-15"  
 output: github\_document  
 ---

Save!

Git people: this is a decent time to commit your edited Rmd.

Render via "Knit HTML" button.

Now revisit the file browser. In addition to foo.Rmd, you should now see foo.md. If there are R chunks that make figures, the usage of markdown output formats will also cause those figure files to be left behind in a sensibly named sub-directory, foo\_files.

If you commit and push foo.md and everything inside foo\_files, then anyone with permission to view your GitHub repo can see a decent-looking version of your report.

If your output format is html\_document, you should still see foo.html. If your output format is github\_document and you see foo.html, that's leftover from earlier experiments. Delete that. It will only confuse you later.

Git people: this is a decent time to commit current state of everything.

## Step 4: Swap out the "guts" of the document

Select everything but the YAML frontmatter and ... delete it!

Write a single English sentence.

Insert an empty R chunk, via the "Chunk" menu in upper right of source editor or with corresponding keyboard shortcut.

```{r}  
## insert your brilliant WORKING code here  
```

Insert 1 to 3 lines of functioning code that begin the task at hand. "Walk through" and run those lines using the "Run" button or the corresponding keyboard shortcut. You MUST make sure your code actually works!

Satisfied? Save!

Now render the whole document via "Knit HTML." Voilà!

Git people: this is a decent time to commit.

## Step 5: Develop your report

In this incremental manner, develop your report. Add code to this chunk. Refine it. Add new chunks. Go crazy! But keep running the code "manually" to make sure it works.

If it doesn't work with you babysitting it, I can guarantee you it will fail, in a more spectacular and cryptic way, when run at arms-length via "Knit HTML" or rmarkdown::render().

Clean out your workspace and restart R and re-run everything periodically, if things get weird. There are lots of chunk menu items and keyboard shortcuts to accelerate this workflow. Render the whole document often to catch errors when they're easy to pinpoint and fix. Save often and commit every time you reach a point that you'd like as a "fall back" position.

You'll develop your own mojo soon, but this should give you your first successful R Markdown experience.

Git people: keep making periodic commits. It's probably better to err on the side of "too often" instead of "too seldom" at this point.

## Step 6: Publish your report

If you've been making HTML, you can put that up on the web somewhere, email to your collaborator, whatever.

No matter what, technically you can publish this report merely by pushing a rendered version to GitHub. However, certain practices make this effort at publishing more satisfying for your audience.

Here are two behaviors I find very frustrating:

* "Here is my code. Behold." This is when someone only pushes their source, i.e. R Markdown or R code AND they want other people to look at their "product". The implicit assumption is that target audience will download code and run it. Sometimes the potential payoff simply does not justify this effort.
* "Here is my HTML. Behold." This is when someone doesn't bother to edit the default output format and accepts HTML only. What am I supposed to do with HTML on GitHub?

Creating, commiting, and pushing markdown is a very functional, lighweight publishing strategy. Use output: github\_document or keep\_md: true if output is html\_document. In both cases, it is critical to also commit and push everything inside foo\_files. Now people can visit and consume your work like any other webpage.

This is (sort of) another example of keeping things machine- and human-readable, which is bliss. By making foo.Rmd available, others can see and run your **actual code**. By sharing foo.md and/or foo.html, others can casually browse your end product and decide if they even want to bother.

### HTML on GitHub

HTML files, such as foo.html, are not immediately useful on GitHub (though your local versions are easily viewable). Visit one and you'll see the raw HTML. Yuck. But there are ways to get a preview: such as <https://rawgit.com> or <http://htmlpreview.github.io>. Expect some pain with HTML files inside private repos. When it becomes vital for the whole world to see proper HTML in its full glory, it's time to use a more sophisticated web publishing strategy.

I have more [general ideas](bit006_github-browsability-wins.html) about how to make a GitHub repo function as a website.

## Troubleshooting

**Make sure RStudio and the rmarkdown package (and its dependencies) are up-to-date.** In case of catastrophic failure to render R Markdown, consider that your software may be too old. R Markdown has been developing rapidly (written 2015-09), so you need a very current version of RStudio and rmarkdown to enjoy all the goodies we describe in this course.

**Get rid of your .Rprofile**, at least temporarily. I have found that a "mature" .Rprofile that has accumulated haphazardly over the years can cause trouble. Specifically, if you've got anything in there relating to knitr, markdown, rmarkdown and RStudio stuff, it may be preventing the installation or usage of the most recent goodies (see above). Comment the whole file out or rename it something else and relaunch or even re-install RStudio.

**Insert a chunk in your .Rmd document so that it renders even when there are errors.** Some errors are easier to diagnose if you can execute specific R statements during rendering and leave more evidence behind for forensic examination. Put this chunk:

near the top of your R Markdown document if you want to soldier on through errors, i.e. turn foo.Rmd into foo.md and/or foo.html no matter what. This is also helpful if you are writing a tutorial and want to demo code that throws an error. You might want to keep this as an [RStudio snippet](https://support.rstudio.com/hc/en-us/articles/204463668-Code-Snippets) for easy insertion.

**Tolerate errors in one specific chunk.** If it's undesirable to globally accept errors, you can still do this for a specific chunk like so:

```{r wing-and-a-prayer, error = TRUE}   
## your sketchy code goes here ;)   
```

**Check your working directory.** It's going to break your heart as you learn how often your mistakes are really mundane and basic. Ask me how I know. When things go wrong consider:

* What is the working directory?
* Is that file I want to read/write actually where I think it is?

Drop these commands into R chunks to check the above:

* getwd() will display working directory at **run time**. If you monkeyed around with working directory with, e.g., the mouse, maybe it's set to one place for your interactive development and another when "Knit HTML" takes over?
* list.files() will list the files in working directory. Is the file you want even there?

**Don't try to change working directory within an R Markdown document**. [Just don't](https://github.com/yihui/knitr/blob/master/FAQ.md). That is all.

**Don't be in a hurry to create a complicated sub-directory structure.** RStudio/knitr/rmarkdown (which bring you the "Knit HTML" button) are rather opinionated about the working directory being set to the .Rmd file's location and about all files living together in one big happy directory. This can all be worked around. But not today.

## Learn More

* [R Markdown Cheat Sheet](https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf)

# Resources

## Where to go from here

R is getting more powerful all the time. There are many exciting features and packages that we have not covered and many more are emerging. Below are a few features I think are worth mentioning:

### Automate testing your code

### Automate your workflow

If you are seasoned programmer and are familiar with the make toolchain, you can use it to automate your workflow too. Professor Jenny Bryan provides [a tutorial](http://stat545.com/automation00_index.html) for her STAT 545 class.

A more research-oriented solution is to use [the remake package](https://github.com/richfitz/remake):

The idea here is to re-imagine a set of ideas from make but built for R. Rather than having a series of calls to different instances of R (as happens if you run make on R scripts), the idea is to define pieces of a pipeline within an R session. Rather than being language agnostic (like make must be), remake is unapologetically R focussed.

remake is still under heavy development and is not yet available on CRAN. You will have to install it from github:

library(devtools)  
install\_github("richfitz/remake")

### Share your work as a R package

If the scripts, data, and/or documents you develop has values to people besides your direct collaborators, it may be worth considering release them as a R package and share it with the world. How cold is that? And it is not hard at all with many helpful functions provided by the devtools package and RStudio.

[R Packages](http://r-pkgs.had.co.nz/) by Hadley Wickham is the best resource for developing R packages. If you want more hands-on tutorials, check out the one by [Prof Jenny Bryan](http://stat545.com/packages00_index.html) and [Chun Chan](http://tinyheero.github.io/jekyll/update/2015/07/26/making-your-first-R-package.html).

### Develop interactive apps with Shiny

Shiny allows you to develop interactive apps that run on top of R. Again Prof Jenny Bryan has a nice [tutorial for getting started with Shiny](http://stat545.com/shiny00_index.html).

### Where to go from here

Practice makes perfection. Use what you learned as much as possible in your research and work.

## Where to find help

1. Help documents
2. Package vignett and manual
3. [R cookbook](http://www.cookbook-r.com/)
4. [stackoverflow](http://stackoverflow.com/questions/tagged/r), A Q&A community for programming
5. Google is your friend
6. Ask a person that may know