PLSC 31101: Computational Tools for Social Science

Rochelle Terman

Fall 2019

Contents

I Before Class **7**

1 Syllabus **9**
- 1.1 Course Description . . . 9
- 1.2 Who should take this course . . . 10
- 1.3 Requirements and Evaluation . . . 10
- 1.4 Activities and Materials . . . 12
- 1.5 Curriculum Outline / Schedule . . . 13

2 Installation **15**
- 2.1 R . . . 15
- 2.2 R Studio . . . 15
- 2.3 R Packages . . . 16
- 2.4 LaTex . . . 16
- 2.5 The Bash Shell . . . 16
- 2.6 Git . . . 17
- 2.7 Other helpful tools . . . 18
- 2.8 Testing your installation . . . 18

3 Homework Rubric **19**

II Course Notes **21**

4 Introduction **23**
- 4.1 The Motivation . . . 23
- 4.2 About This Class . . . 27
- 4.3 Learning How to Program . . . 29

5 R Basics **35**
- 5.1 What is R? . . . 35
- 5.2 RStudio . . . 36
- 5.3 R Packages . . . 41
- 5.4 R Markdown . . . 43

6 R Syntax **49**
6.1 Variables . . . 49
6.2 Functions . . . 52
6.3 Data Types . . . 55

7 Data Classes and Structures **59**
7.1 Vectors . . . 60
7.2 Lists . . . 64
7.3 Factors . . . 68
7.4 Matrices . . . 71
7.5 Dataframes . . . 73

8 Subsetting **79**
8.1 Subsetting Vectors . . . 79
8.2 Subsetting Lists . . . 83
8.3 Subsetting Matrices . . . 86
8.4 Subsetting Dataframes . . . 87
8.5 Sub-assignment . . . 91

9 Project Workflow **97**
9.1 Organizing Code . . . 97
9.2 Importing and Exporting . . . 102

10 Data Transformation **107**
10.1 Introduction to Data . . . 107
10.2 Introduction to Tidyverse . . . 111
10.3 `dplyr` Functions . . . 112
10.4 Challenges . . . 123

11 Tidying Data **125**
11.1 Wide vs. Long Formats . . . 125
11.2 Tidying the Gapminder Data . . . 128
11.3 `tidyr` Functions . . . 128
11.4 More `tidyverse` . . . 132
11.5 Challenges . . . 133

12 Relational Data **135**
12.1 Why Relational Data . . . 135
12.2 Keys . . . 137
12.3 Joins . . . 138
12.4 Defining Keys . . . 139
12.5 Duplicate Keys . . . 141
12.6 Challenges . . . 141

13 Plotting **143**
13.1 The Dataset . . . 143
13.2 R Base Graphics . . . 143

13.3 ggplot2 . . . 153
13.4 Saving plots . . . 166

14 Statistical Inferences **167**
14.1 Statistical Distributions . . . 167
14.2 Inferences and Regressions . . . 170

15 Strings and Regular Expressions **185**
15.1 String Basics . . . 185
15.2 Regular expressions . . . 189
15.3 Common Tools . . . 192
15.4 Other types of patterns . . . 199
Acknowledgments . . . 200

16 Programming in R **201**
16.1 Conditional Flow . . . 201
16.2 Functions . . . 205
16.3 Iteration . . . 212

17 Collecting Data from the Web **221**
17.1 Introduction . . . 221
17.2 Web APIs . . . 221
17.3 Collecting Twitter Data with RTweet . . . 226
17.4 Writing API Queries . . . 229
17.5 Webscraping . . . 235
17.6 Scraping Presidential Statements . . . 241

18 Text Analysis **247**
18.1 Preprocessing . . . 247
18.2 Sentiment Analysis and Dictionary Methods . . . 255
18.3 Distinctive Words . . . 258
18.4 Structural Topic Models . . . 263

19 Git and Github **273**
19.1 Learning Objectives . . . 273
19.2 Starting with Git . . . 274
19.3 Collaborating . . . 282

Part I

Before Class

Chapter 1

Syllabus

- Instructor: Rochelle Terman, rterman@uchicago.edu
- TA: Pete Cuppernull, pcuppernull@uchicago.edu
- Time: Tuesdays and Thursdays, 12:30 pm – 1:50 pm
- Place: Cobb Hall 303
- Office hours:
  - Rochelle Terman: Tuesdays, 2:30-4:30, Pick 411. Sign up here
  - Pete Cuppernull: Mondays, 2:00-4:00, Gates-Blake 211

1.1 Course Description

The purpose of this course is to provide graduate students with the critical computing skills necessary to conduct research in quantitative / computational social science. This course is not an introduction to statistics, computer science, or specialized social science methods. Rather, the focus will be on practical skills necessary to be successful in further methods work. The first portion of the class introduces students to basic computer literacy, terminologies, and the R programming language. The second part of the course provides students the opportunity to use the skills they learned in part 1 towards practical applications such as webscraping, data collection through APIs, automated text analysis, etc. We will assume no prior experience with programming or computer science.

Objectives

By the end of the course, students should be able to:

- Understand basic programming terminologies, structures, and conventions.
- Write, execute, and debug R code.
- Produce reproducible analyses using R Markdown.
- Clean, transform, and wrangle data using the `tidyverse` packages.
- Scrape data from websites and APIs.
- Parse and analyze text documents.
- Be familiar with the concepts and tools of a variety of computational social science applications.
- Master basic Git and GitHub workflows
- Learn independently and train themselves in a variety of computational applications and tasks through online documentation.

1.2 Who should take this course

This course is designed for Political Science Graduate students, but any graduate student is welcome. We will not presume any prior programming or computer science experience.

1.3 Requirements and Evaluation

Final Grades

This is a graded class based on the following:

- Completion of assigned homework (50%)
- Participation (25%)
- Final project (25%)

Assignments

Assignments will be assigned at the end of every Thursday session. They will be due one week later, unless otherwise noted. The assignments are intended to expand upon the lecture material and to help students develop the actual skills that will be useful for applied work. The assignments will be frequent but each of them should be fairly short.

You are encouraged to work in groups, but the work you turn in must be your own. Group submission of homework, or turning in copies of the same code or output, is not acceptable. While you are encouraged to use the internet to help you debug, but do not copy and paste large chunks of code that you do not understand. Remember, the only way you actually learn how to write code is to write code!

Portions of the homework in `R` should be completed using R markdown, a markup language for producing well-formatted documents with embedded R code and outputs. To submit your homework, knit the R Markdown file to PDF, and then submit the PDF file through Canvas (unless otherwise noted).

Class Participation

The class participation portion of the grade can be satisfied in one or more of the following ways:

- attending the lectures
- asking and answering questions in class
- attending office hours
- contributing to class discussion through the Canvas site, and/or
- collaborating with the computing community, either by attending a workshop or meetup, submitting a pull request to a github repository (including the class repository), answering a question on StackExchange, or other involvement in the social computing / digital humanities community.

Final Project

Students have two options for class projects:

1. **Data project**: Use the tools we learned in class on your own data of interest. Collect and/or clean the data, perform some analysis, and visualize the results. Post your reproducible code on Github.
2. **Tool project**: Create a tutorial on a tool we didn't cover in class. Ideas include: bash, LaTex, pandoc, quanteda, tidytext, etc. Post it on github.

Students are required to write a short proposal by **November 7** (no more than 2 paragraphs) in order to get approval / feedback from the instructors.

Project materials (i.e. a github repo) will be due by end of day on **December 9**. We will specify submission details in class.

On **December 10** we will have a **lightning talk session** where students can present their projects in a maximum 5 minute talk.

Late Policy and Incompletes

All deadlines are strict. Late assignments will be dropped a full letter grade for each 24 hours past the deadline. Exceptions will be made for students with a documented emergency or illness.

I will only consider granting incompletes to students under extreme personal/family duress.

Academic Integrity

I follow a zero-tolerance policy on all forms of academic dishonesty. All students are responsible for familiarizing themselves with, and following, university policies regarding proper student conduct. Being found guilty of academic dishonesty is a serious offense and may result in a failing grade for the assignment in question, and possibtire course.

1.4 Activities and Materials

Course Structure

Classes will follow a "workshop" style, combining lecture, demonstration, and coding exercises. We envision the class to be as interactive / hands on as possible, with students programming every session. **You must bring a laptop to class.**

It is important that students **complete the requisite reading** before class. I anticipate spending 1/2 the class lecturing, and 1/2 practicing with code challenges.

Course Website

All course notes will be posted on https://plsc-31101.github.io/course/, including class notes, code demonstrations, sample data, and assignments. Students will be assigned reading from these notes before every class.

Students are encouraged to submit pull requests to the website repository, for example if they find a typo, or if they found a particularly helpful resource that would aid other students.

Students projects will also be shared on the course website.

Canvas

We will use Canvas for communication (announcements and questions) and turning in assignments.

You should ask questions about class material and assignments through the Canvas website so that everyone can benefit from the discussion. We encourage you to respond to each other's questions as well. Questions of a personal nature can be emailed to us directly.

Computer Requirements and Software

By the end of the first week, you should install the following software on your computer:

- Access to the UNIX command line (e.g., a Mac laptop, a Bash wrapper on Windows)
- Git
- R and RStudio (latest versions)

This requires a computer that can handle all this software. Almost any Mac will do the job. Most Windows machines are fine too if they have enough space and memory.

See Install Page for more information. We will be having an **InstallFest on Wednesday, Oct 2 from 9:30 to 11:30 in Pick 504**. for those students experiencing difficulties downloading and installing the requisite software.

1.5 Curriculum Outline / Schedule

1. **Oct 1** - Introduction
    - course objectives, logistics, overview of programming and reproducible research.
2. **Oct 3** - R Tools
    - R Studio, R Markdown, packages, help.
3. **Oct 8** - R Syntax
    - basic syntax, variables, functions, data types.
4. **Oct 10** - Data Structures
    - vectors, lists, factors, matrices, data frames.
5. **Oct 15** - Indexing and Subsetting
    - subsetting formats, operators, boolean conditionals, sub-assignment, applications.
6. **Oct 17** - Introduction to Data
    - common terms, formats, tidy data, storage, import/export, exploratory data analysis.
7. **Oct 22** - Manipulating data with `dplyr`
    - importing data, subsetting, summarizing, and conducting calculations across groups, piping.

8. **Oct 24** - Tidying data with `tidyr`
    - tidy data principles, gather, spread, separate, unite
9. **Oct 29** - Merging and Linking Data
    - relational data, keys, joins, missing values.
10. **Oct 31** - Visualization
    - base plotting, ggplot, grammar of graphics, writing images.
11. **Nov 5** - Hypothesis testing and regressions
    - models, model objects, stargazer.
12. **Nov 7** - Strings and Regex
13. **Nov 12** - R programming 1
    - conditional flow, functions.
14. **Nov 14** - R Programming 2
    - iterations, map.
15. **Nov 19** - Collecting data with APIs
    - requests, RESTful APIs, queries, API libraries.
16. **Nov 21** - Webscraping
    - HTML, CSS, In-Browser tools, scraping tools.
17. **Nov 26** - Text analysis 1
    - preprocessing, DTM, dictionary methods, distinctive words.
18. **Nov 28** - No class, Thanksgiving
19. **Dec 3** - Text analysis 2
    - supervised vs. unsupervised learning, Vector-space models, topic models.
20. **Dec 5** - Git / Github
21. **Dec 10, 1:30-3:30pm** - Final project lightning talks

Chapter 2

Installation

To participate in PLSC 31101, you will need access to the software described below.

Once you've installed all of the software below, test your installation by following the instructions at the bottom on this page.

2.1 R

R is a programming language that is especially powerful for data exploration, visualization, and statistical analysis.

To download R, go to CRAN (the **C**omprehensive **R** **A**rchive **N**etwork).

A new major version of R comes out once a year, and there are 2-3 minor releases each year. It's a good idea to update regularly. Upgrading can be a bit of a hassle, especially for major versions, which require you to reinstall all your packages, but putting it off only makes it worse.

2.2 R Studio

To interact with R, we use RStudio. Please install the latest desktop version of RStudio IDE. We will not support RStudio cloud.

2.3 R Packages

You'll also need to install some R packages. An R package is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R.

Many of the packages that you will learn in this class are part of the so-called `tidyverse`. The packages in the `tidyverse` share a common philosophy of data and R programming, and are designed to work together naturally.

You can install the complete tidyverse with a single line of code in R Studio:

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

On your own computer, type that line of code in the RStudio console, and then press enter to run it. R will download the packages from CRAN and install them on to your computer. If you have problems installing, make sure that you are connected to the internet, and that https://cloud.r-project.org/ isn't blocked by your firewall or proxy.

We will also be requiring a few other important packages. Run the following line of code to download all of them at once:

```
install.packages("rmarkdown", "knitr", "gapminder",
                 "stargazer", "rtweet", "kableExtra",
                 "nycflights13", "devtools", "tm", "wordcloud",
                 "matrixStats", "SnowballC", "tidytext",
                 "textdata", "stm")
```

2.4 LaTex

In order to knit rmarkdown files to pdf files, you need to install some version of Latex. For students who have not installed LaTeX before, we recommend that you install TinyTeX (https://yihui.name/tinytex/).

Open RStudio and type these lines into the command-line console:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

2.5 The Bash Shell

Bash is a commonly-used shell that gives you the power to do simple tasks more quickly.

Windows

Install Git for Windows by downloading and running the installer. This will provide you with both Git and Bash in the Git Bash program. **NOTE**: on the ~6th step of installation, you will need to select the option "Use Windows' default console window" rather than the default of "Use MinTTY" in order for nano to work correctly.

After the installer does its thing, it leaves the window open, so that you can play with the "Git Bash".

Chances are that you want to have an easy way to restart that Git Bash. You can install shortcuts in the start menu, on the desktop or in the QuickStart bar by calling the script /share/msysGit/add-shortcut.tcl (call it without parameters to see a short help text).

Mac OS X

The default shell in all versions of Mac OS X is bash, so no need to install anything. You access bash from the Terminal (found in `/Applications/Utilities`). You may want to keep Terminal in your dock for this class.

Linux

The default shell is usually Bash, but if your machine is set up differently you can run it by opening a terminal and typing bash. There is no need to install anything.

2.6 Git

Git is a version control system that lets you track who made changes to what when and has options for easily updating a shared or public version of your code on github.com. You will need a supported web browser (current versions of Chrome, Firefox or Safari, or Internet Explorer version 9 or above).

Windows

Git should be installed on your computer as part of your Bash install (described above).

Mac OS X

For OS X 10.9 and higher, install Git for Mac by downloading and running the most recent "mavericks" installer from this list. After installing Git, there will not be anything in your `/Applications` folder, as Git is a command line program. **For older versions of OS X (10.5-10.8)** use the most recent available installer labelled "snow-leopard" available here.

Linux

If Git is not already available on your machine you can try to install it via your distro's package manager. For Debian/Ubuntu run sudo apt-get install git and for Fedora run sudo yum install git.

2.7 Other helpful tools

While not required, I recommend you install the following tools:

1. Google Chrome is an up-to-date web browser.
2. Sublime Text is a free, advanced text editor.

2.8 Testing your installation

If you have trouble with installation, please come to the Installfest on **Wed Oct 2, 10-12 in Pick 411**.

Open a command line window ('terminal' or, on windows, 'git bash'), and enter the following commands (without the $ sign):

```
$ R --version
$ git --version
```

If everything has been installed correctly, those commands *should* print output version information.

NB: If you're using git bash, the `R --version` command may not work. In this case, just make sure you can open up RStudio.

Software Carpentry maintains a list of common issues that occur during installation may be useful for our class here: Configuration Problems and Solutions wiki page.

Credit: Thanks to Software Carpentry for providing installation guidelines.

Chapter 3

Homework Rubric

Each question will be graded along the following criteria:

Check Minus: The code doesn't run, or does not do what the question is asking.

Check: The code works, but exhibits one of the following problems:

1) poor documentation / disorganized;
2) repetitive / redundant / overly complex;
3) graphs and tables lack labels or are otherwise difficult to understand.

Check Plus: The code works, and is:

1) well-documented and organized;
2) clean and efficient;
3) graphs and tables have proper labels and are easily readable.

Part II

Course Notes

Chapter 4

Introduction

4.1 The Motivation

Here's the dream:

....

> Computers have revolutionized research, and that revolution is only beginning. Every day, social scientists and humanists all over the world use them to study things that are too big, too small, too fast, too slow, too expensive, too dangerous, or just too hard to study any other way.

Now here's the reality

....

> Every day, scholars all over the world waste time wrestling with computers. Tasks that should take a few moments take hours or days, and many things never work at all. When scholars try to get help, they are inundated with unhelpful information, and give up.

This sorry state of affairs persists for three reasons:

1. **No room, no time.** Everybody's schedule is full —- there's simply not space to add more about computing without dropping something else.
2. **The blind leading the blind.** The infrastructure doesn't exist to help scholars develop the skills they need. Senior researchers can't teach the next generation how to do things that they don't know how to do themselves.
3. **Autodidact Chauvinism.** Since there are no classrooms, scholars are pressured to teach themselves. But self-learning is time consuming and

nearly impossible without a base level of knowledge.

Despite these challenges, there are great reasons to learn how to program:

1. Practical Efficiency

Even though it takes some time at first, learning how to program can save you an enormous amount of time doing basic tasks that you would otherwise do by hand, once you get the hang of it.

2. New Tools

Some things are impossible, or nearly impossible to do by hand. Computers open the door for new tools and methods, but many require programming skills.

3. New Data

The Internet is a wealth of data, waiting to be analyzed! Whether its collecting Twitter data, working with the Congress API, or scraping websites, programming knowledge is a must.

4. Better Scholarship

(Quality) programming can open the door to better transparency, reproducibility, and collaboration in the Social Sciences and the Humanities.

Goal of the Class: Learn to Learn

> The basic learning objective of this course is to leave here with the knowledge and skills required to learn on your own, whether that's through programming documentation, StackExchange and other online fora, courses or D-Lab workshops.

By the end of the course, students should be able to:

- Understand basic programming terminologies, structures, and conventions.
- Write, execute, and debug R code.
- Produce reproducible analyses using R Markdown.
- Clean, transform, and wrangle data using the `tidyverse` packages.
- Scrape data from websites and APIs.
- Parse and analyze text documents.
- Be familiar with the concepts and tools of a variety of computational social science applications.
- Master basic Git and GitHub workflows
- **Learn independently and train themselves in a variety of computational applications and tasks through online documentation.**

This Course will not…

- teach you to be a professional programmer or software developer

- teach you statistics, computer science, or specialized social science / digital humanities methods

Why not just take a Computer Science course?

Computer science courses don't anticipate the types of questions social scientists might ask, and therefore they - introduce many unnecessary concepts - do a poor job of explaining how computer programming tools might be used by social scientists - are too resource intensive for the average social scientist

Programming is not just use useful for computer scientists, methodologists, or people who work with "big data."

A Practical example

To illustrate, here's a practical example that comes out of my own research:

Task 1 (by hand)

Topic: International Human Rights "Naming and Shaming"

Question: Who Shames Whom on What?

Data: UN Human Rights Committee's Universal Periodic Review

From Antigua & Barbuda 2011 review:

1. Continue with the efforts to prevent, punish and eradicate all forms of violence against women (Argentina);

2. Accede to the Second Optional Protocol to the International Covenant on Civil and Political Rights, aimed at abolishing the death penalty, and take all necessary steps to remove the death penalty from Antigua and Barbuda law (Australia);

3. Improve conditions in Antigua and Barbuda's prisons and detention facilities (Australia);

The Task: Parse a bunch of reports (PDFs) into a dataset (CSV). Add meta-data for issue, action, sentiment.

From	To	Text	Action	Sentiment	Issue	Institution	Response
Argentina	Antigua & Barbuda	Continue with the efforts to prevent, punish and eradicate all forms of violence against women	continue	2	Women's rights		support
Australia	Antigua & Barbuda	Accede to the Second Optional Protocol to the International Covenant on Civil and Political Rights, aimed at abolishing the death penalty, and take all necessary steps to remove the death penalty from Antigua and Barbuda law	accede	5	Death penalty,International instruments	ICCPR, OP	reject
Australia	Antigua & Barbuda	Improve conditions in Antigua and Barbuda's prisons and detention facilities	improve	4	Detention		support

How much time will it take?

By hand: 40,000 recommendations x 3 min per recommendation x 8-hour days x 5-day weeks = **12 months**

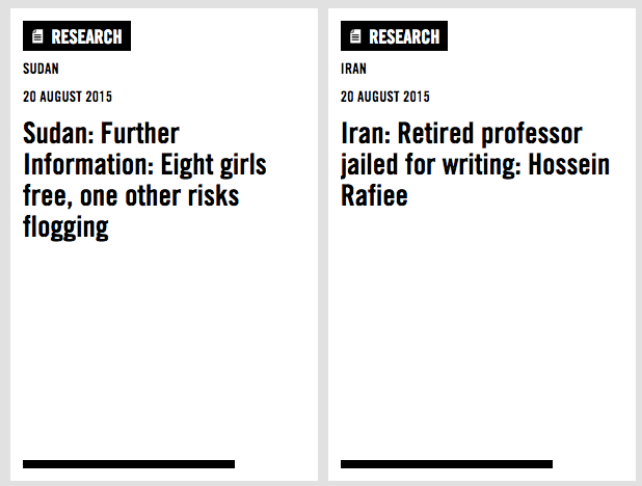
Task 2 (by hand)

What if we wanted to extend this research?

Question: How does UPR shaming compare to actual human rights abuses?

Data: Amnesty International's Urgent Actions

Task: Collect all of Amnesty International's Urgent Actions, add metadata for issue, country.



How much time will it take?

By hand: 25,000 recommendations x 3 min per recommendation x 8-hour days x 5-day weeks = **7.5 months**

Tasks 1 & 2 (with a computer)

With a computer, we can write a program that:

1. Parses recommendations into a CSV.
2. Codes recommendations by issue, action, sentiment using computational text analysis tools.
3. Uses webscraping to collect all of Amnesty International's urgent actions.
4. Run simple regression models with R to correlate Amnesty reports with UPR shaming.

Total time: 2 months

Time Saved: 1.5 years

4.2 About This Class

About Me

My name is Rochelle Terman and I'm a faculty member in Political Science.

- A few years ago, I didn't know how to program. Now I program almost every day.
- I program mostly in Python and R. I have a special interest in text analysis and webscraping.
- My substantive research is on international norms and human rights.
- I won't be able to answer all your questions.
- No one will.
- But especially me.

Course Structure

The course is divided into two main sections:

1. Skills

Basic computer literacy, terminologies, and programming languages:

- Base R: objects and data structures.
- `tidyverse` for data analysis.
- Modeling and visualization.
- Key programming concepts (iteration, functions, conditional flow, etc).

We're using `R` because it's the common programming language among Political Scientists. But if you understand the *concepts*, you should able to pick up Python and other languages pretty easily.

2. Applications

Use the skills they learned in part 1 towards practical applications:

- Webscraping.
- APIs.
- Computational Text Analysis.
- Version control and communication.

The goal is to **introduce** the students to a medley of common applications so that they can discover which avenue to pursue in their own research, and what such training would entail.

Class Activities

Classes will follow a "workshop" style, combining lecture, demonstration, and coding exercises. We envision the class to be as interactive / hands on as possible, with students programming every session. **You must bring a laptop to class.**

It is important that students **complete the requisite reading** before class. I anticipate spending 1/2 the class lecturing, and 1/2 practicing with code challenges.

Course Websites

Class notes and other materials are available here: https://github.com/plsc-31101/course/

We will also be using Canvas to distribute/accept assignments, and for discussion. Please use the discussion forums liberally.

Evaluation

This is a graded class based on the following: - Completion of assigned homework (50%) - Participation (25%) - Final project (25%)

If you want to audit, please let me know ASAP.

Assignments

- In general, assignments are assigned at the end of lecture, and due the following week.
- Exceptions will be noted.
- The first assignment is due next week, before class on Thursday Oct 10. It is on Canvas.
- Turn in assignments on Canvas.
- Work in groups, but submit your own.

Participation

The class participation portion of the grade can be satisfied in one or more of the following ways:

- attending the lectures,
- asking and answering questions in class,
- attending office hours,
- contributing to class discussion through the Canvas/Piazza site,
- collaborating with the computing community.

Final Project

Students have two options for class projects:

1. *Data project*: Using the tools we learned in class on your own data of interest.

2. *Tutorial project*: Create a tutorial on a tool we didn't cover in class.

Both options require an R markdown file narrating the project.

Students are required to write a short proposal by **November 7** (no more than 2 paragraphs) in order to get approval / feedback from the instructors.

Project materials (i.e. a github repo) will be due by end of day on **December 9**. We will specify submission details in class.

On **December 10** we will have a **lightning talk session** where students can present their projects in a maximum 5 minute talk.

Software

- Installation instructions are onthe website.
- Get started **EARLY**.
- Then go to the Installfest (Wednesday, Oct 2, 9:30-11:30 in Pick 504) to double check your installation.
- If you have computer troubles, post the problem on the discussion forums, with as much detail as possible.

4.3 Learning How to Program

Before we talk about what it takes to learn how to program, let's first review what programming is.

What is Programming?

A *program* is a sequence of instructions that specifies how to perform a computation. Most programs are written in a human-readable *programming language* (or "source code") and then executed with the help of a *compiler* or *interpreter*.

A few basic instructions appear in just about every language:

1. **input**: Get data from the keyboard, a file, the network, or some other device.
2. **output**: Display data on the screen, save it in a file, send it over the network, etc.
3. **math**: Perform basic mathematical operations like addition and multiplication.
4. **conditional execution**: Only perform tasks if certain conditions are met.
5. **iteration**: Do the same task over and over again on different inputs.

That being said, programming languages differ from one another in the following ways:

1. **Syntax**: whether to add a semicolon at the end of each line, etc.
2. **Usage**: JavaScript is for building websites, R is for statistics, Python is general purpose, etc.
3. **Level**: how close you are to the hardware. 'C' is often considered to be the lowest (or one of the lowest) level languages.
4. **Object-oriented:** "objects" are data + functions. Some programming languages are object-oriented (e.g. Python) and some aren't (e.g. C).
5. **Many more**: Here's a list of all the types of programming languages out there.

What Language Should You Learn?

Most programmers can program in more than one language. That's because they know *how to program* generally, as opposed to "knowing" Python, R, Ruby, or whatever.

So what should your first programming language be? That is, what programming language should you use to learn *how to program*? At the end of the day, the answer depends on what you want to get out of programming. Many people recommend Python because it's fun, easy, and multi-purpose. Here's an article that can offer more advice.

In this class, we'll be using R because it's the most popular language in our disciplnary community (of Political Scientists.)

Regardless of what you choose, you will probably grow comfortable in one language while learning the basic concepts and skills that allow you to 'hack' your

way into other languages. That's because **programming is an extendible skill.**

Thus "knowing how to program" means learning how to *think* like a programmer, not necessarily knowing all the language-specific commands off the top of your head. **Don't learn specific programming languages; learn how to program.**

What Programming is Really Like

Here's the sad reality: When you're programming, 80% or more of your time will be spent debugging, looking stuff up (like program-specific syntax, documentation for packages, useful functions, etc.), or testing. This does not just apply to beginner or intermediate programmers, although you will grow more "fluent" over time.

> Google software engineers write an average of 10-20 lines of code per day. **The Lesson**: Programming is a slow activity, especially in the beginning.

If you're a good programmer, you're a good detective!

How to Learn

Here are some tips on how to learn computer programming:

1. Learning to program is 5% intelligence, 95% endurance.
2. Like learning to play an instrument or speak a foreign language, it takes practice, practice, practice.
3. Program a little bit every day.
4. Program with others. Do the problem sets in pairs or groups.
5. It's better to type than to copy and paste.

6. Most "programming" is actually researching, experimenting, thinking.
7. Stay organized.

The 15 minute rule

15 min rule: when stuck, you HAVE to try on your own for 15 min; after 15 min, you HAVE to ask for help.- Brain AMA pic.twitter.com/MS7FnjXoGH

— Rachel Thomas (**?**) August 14, 2016

We will follow the **15 minute rule** in this class. If you encounter a problem in your assignments, spend 15 minutes troubleshooting the problem on your own. After 15 minutes, if you still cannot solve the problem, **ask for help**.

(Hat tip to Computing for Social Sciences)

Debugging

Those first 15 minutes should be spent trying to debug your code. Here are some tips:

- Read the errors!
- Read the documentation
- Make it smaller
- Figure out what changed
- Check your syntax
- Print statements are your friend

Using the Internet

You should also make use of Google and StackOverflow to resolve the error. Here's some tips for how to google errors:

- google: name-of-program + text in error message.
- Remove user- and data-specific information first!
- See if you can find examples that do and don't produce the error. Try other people's code, but don't fall into the copy-paste trap.

Asking for Help

We will use Canvas/Piazza for class-related questions and discussion. You are highly encouraged to ask questions, and answer one another's questions.

1. Include a brief, informative title.
2. Explain what you're trying to do, and how it failed.

3. Include a reproducible example.

Here are some helpful guidelines on how to properly ask programming questions:

1. "How to Ask Programming Questions," ProPublica
2. "How do I ask a good question?" StackOverflow
3. "How to properly ask for help" Computing for Social Science

Chapter 5

R Basics

This unit introduces you to the R programming language and the tools we use to program in R. We will explore:

1. **What is R?**, a brief introduction to the R language.
2. **RStudio**, a tour of the Interactive Development Environment RStudio.
3. **R Packages**, extra tools and functionalities.
4. **R Markdown**, a type of R script file we will be working with in this class.

5.1 What is R?

R is a versatile, open-source programming and scripting language that is useful both for statistics and data science. It is inspired by the programming language `S`. Some of its **best features** are:

- It is free, open-source, and available on every major platform. As a result, if you do your analysis in R, most people can easily replicate it.
- It contains a massive set of packages for statistical modelling, machine learning, visualization, and importing and manipulating data. Over 14,000 packages are available as of August 2019. Whatever model or graphic you are trying to do, chances are that someone has already tried to do it (and a package for it exists).
- It is designed for statistics and data analysis, but also general-purpose programming.
- It is an Interactive Development Environment tailored to the needs of interactive data analysis and statistical programming.

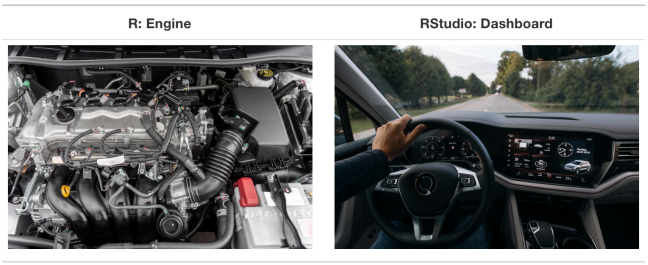
- It has powerful tools for communicating your results. R packages make it easy to produce HTML or PDF reports, or create interactive websites.
- A large and growing community of peers.

R also has a number of **shortcomings**:

- It has a steeper learning curve than SPSS or Stata.
- R is not a particularly fast programming language, and poorly written R code can be terribly slow. R is also a profligate user of memory.
- Much of the R code you will see in the wild is written in haste to solve a pressing problem. As a result, code is not very elegant, fast, or easy to understand. Most users do not revise their code to address these shortcomings.
- Inconsistency is rife across contributed packages, even within base R. You are confronted with over 20 years of evolution every time you use R. Learning R can be tough, because there are many special cases to remember.

5.2 RStudio

Throughout this class, we will assume that you are using R via RStudio. First-time users often confuse the two. At its simplest, R is like a car's engine, while RStudio is like a car's dashboard.



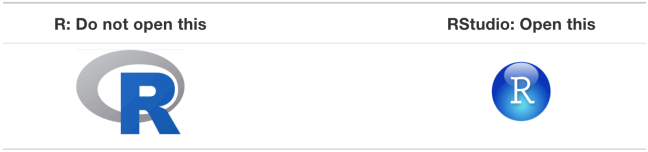
More precisely, R is a programming language that runs computations, while RStudio is an *integrated development environment (IDE)* that provides an interface with many convenient features and tools. Just as the way of having access to a speedometer, rear-view mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

RStudio includes a console, a syntax-highlighting code editor, as well as tools for plotting, history, debuggingm and workspace management. It is also free and open-source. Yay!

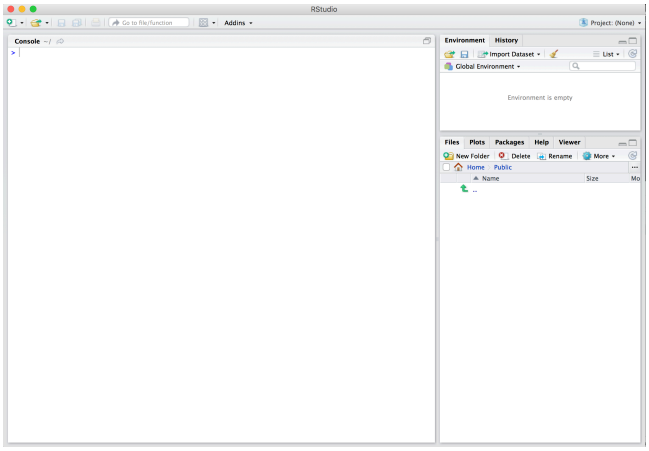
> NB: We do not have to use RStudio to use R. For example, we can write R code in a plain text editor (like `textedit` or `notepad`) and

then execute the script using the shell (e.g. `terminal` in Mac). But this is not ideal.

After you install R and RStudio on your computer, you will have two new applications you can open. We will always work in RStudio – not in the R application.



After you open RStudio, you should see something similar to this:



5.2.1 Console

There are two main ways of interacting with R: by using the **console** or by using the **script editor**.

The console window (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do and where it will show the results of a command.

You can type commands directly into the console, but they will be forgotten when you close the session. Try it out now.

```
> 2 + 2
```

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing, copy-pasting, or sending from the script editor using `Ctrl-Enter`), R will try to execute it andm when ready, show the results and come back with a new `>`-prompt to wait for new commands.

If R is still waiting for you to enter more data because it is not complete yet, the console will show a `+` prompt. It means that you have not finished entering a complete command. This happens when you have not 'closed' a parenthesis or quotation. If you are in RStudio and this happens, click inside the console window and press `Esc`; this should help get you out of trouble.

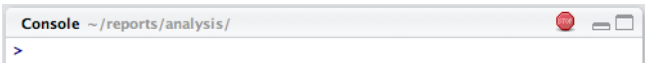
```
> "This is an incomplete quote
+
```

More Console Features

1. Retrieving previous commands: As you work with R, you will often want to re-execute a command which you previously entered. Recall previous commands using the up and down arrow keys.

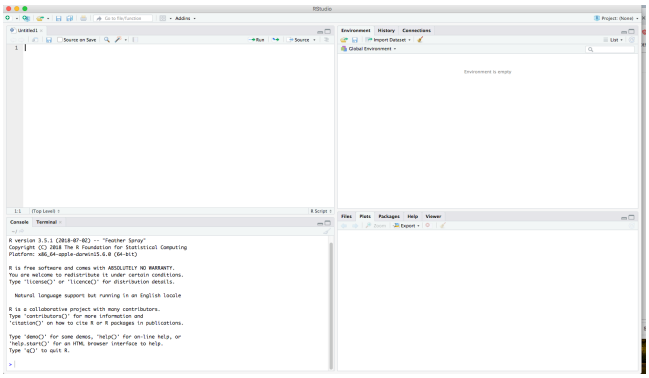
2. Console title bar: This screenshot illustrates a few additional capabilities provided by the console title bar:

   - Display of the current working directory.
   - The ability to interrupt R during a long computation.
   - Minimizing and maximizing the console in relation to the Source pane (by using the buttons at the top-right or by double-clicking the title bar).



5.2.2 Scripts

It is better practice to enter commands in the script editor and save the script. This way, you have a complete record of what you did, you can easily show others how you did it, and you can do it again later on if needed. Open it up either by clicking the *File* menu and selecting *New File*, then R script; or by using the keyboard shortcut Cmd/Ctrl + Shift + N. Now you will see four panes.



The script editor is a great place to put code you care about. Keep experimenting in the console, but, once you have written code that works and does what you want, put it in the script editor.

RStudio will automatically save the contents of the editor when you quit RStudio and load them when you re-open RStudio. Nevertheless, it is a good idea to save your scripts regularly and to back them up.

5.2.3 Running Code

While you certainly can copy-paste code that you would like to run from the editor into the console, this workflow is pretty inefficient. The key to using the script editor effectively is to memorize one of the most important keyboard shortcuts in RStudio: **`Cmd/Ctrl + Enter`**. This executes the current R expression from the script editor in the console.

For example, take the code below. If your cursor is somewhere on the first line, pressing `Cmd/Ctrl + Enter` will run the complete command that generates `dems`. It will also move the cursor to the next statement (beginning with `reps`). That makes it easy to run your complete script by repeatedly pressing `Cmd/Ctrl + Enter`.

```
dems <- (55 + 70) * 1.3

reps <- (20 - 1) / 2
```

Instead of running expression by expression, you can also execute the complete script in one step: `Cmd/Ctrl + Shift + S`. Doing this regularly is a great way to check that you have captured all the important parts of your code in the script.

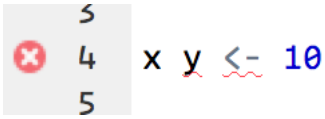
5.2.4 Comments

Use # signs to add comments within your code chunks. You are encouraged to regularly comment within your code. Anything to the right of a # is ignored by R. Each line of a comment needs to begin with a #.

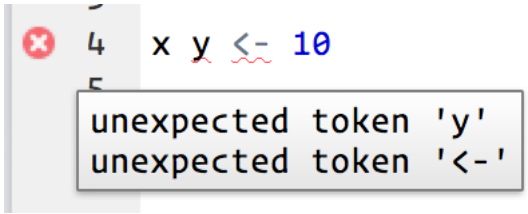
```
# This is a comment.
# This is another line of comments.
```

5.2.5 Diagnostics and errors

The script editor will also highlight syntax errors with a red squiggly line and a cross in the sidebar:



You can hover over the cross to see what the problem is:



If you try to execute the code, you will see an error in the console:

```
> x y <- 10
Error: unexpected symbol in "x y"
>
```

When errors happen, your code is haulted – meaning it is never executed. Errors can be frustrating in R, but, with practice, you will be able to debug your code quickly.

5.2.6 Errors, Messages, and Warnings

One thing that intimidates new R and RStudio users is how it reports errors, warnings, and messages. R reports errors, warnings, and messages in a glaring font, which makes it seem like it is scolding you. However, seeing red text in the console is not always bad:

1. **Errors**: When the text is a legitimate error, it will be prefaced with "Error:", and R will try to explain what went wrong. Generally, when there is an error, the code will not run. *Think of errors as a red traffic light: something is wrong!*

2. **Warnings**: When the text is a warning, it will be prefaced with "Warning:", and R will try to explain why there is a warning. Generally, your code will still work, but perhaps not in the way you would expect. *Think of warnings as a yellow traffic light: everything is working fine, but watch out/pay attention.*

3. **Messages**: When the text doesn not start with either "Error:" or "Warning:", it is just a friendly message. These are helpful diagnostic messages and they do not stop your code from working. *Think of messages as a green traffic light: everything is working fine, and keep on going!*

5.2.7 R Environment

Turn your attention to the upper right pane. This pane displays your "global environment" and contains the data objects you have saved in your current session. Notice that we have the two objects created earlier, `dems` and `reps`, along with their values.

You can list all objects in your current environment by running:

```
ls()
#> [1] "dems" "reps"
```

Sometimes we want to remove objects that we no longer need.

```
x <- 5
rm(x)
```

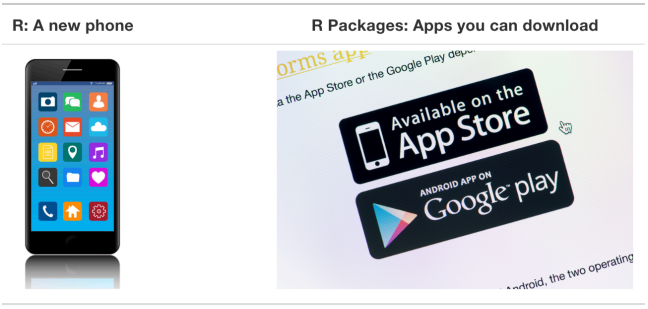
If you want to remove all objects from your current environment, you can run:

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

5.3 R Packages

The best part about R are its user-contributed packages (also called "libraries"). A *package* is a collection of functions (and sometimes data) that can be used by other programers.

A good analogy for R packages is they are like apps you can download onto a mobile phone:



So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it does not have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.

Let's continue this analogy by considering the Instagram app for editing and sharing pictures. Say you have purchased a new phone and you would like to share a photo you have just taken with friends and family on Instagram. You need to:

1. **Install the app**: Since your phone is new and does not include the Instagram app, you need to download the app from either the App Store or Google Play. You do this once and you are set for the time being. You might need to do this again in the future when there is an update to the app.
2. **Open the app**: After you have installed Instagram, you need to open the app.

The process is very similar for using an R package. You need to:

1. **Install the package**: This is like installing an app on your phone. Most packages are not installed by default when you install R and RStudio. Thus, if you want to use a package for the first time, you need to install it first. Once you have installed a package, you likely will not install it again unless you want to update it to a newer version.
2. **"Load" the package**: Loading a package is like opening an app on your phone. Packages are not loaded by default when you start RStudio on your computer; you need to load each package you want to use every time you start RStudio.

5.3.1 Installing Packages

First, we download the package from one of the CRAN mirrors onto our computer. For this we use `install.packages("package-name")`. If you have not set a preferred CRAN mirror in your `options()`, a menu will pop up asking you to choose a location.

Let's download the package `dplyr`.

```
install.packages("dplyr")
```

If you run into errors later in the course about a function or package not being found, run the `install.packages` function to make sure the package is actually installed.

Important: Once we download the package, we never need to run `install.packages` again (unless we get a new computer).

5.3.2 Loading Packages

Once we download the package, we need to load it into our session to use it. This is required at the beginning of each R session. This step is necessary because, if we automatically loaded every package we have ever downloaded, our computer would fry.

```
library(dplyr)
```

The message tells you which functions from `dplyr` conflict with functions in base R (or from other packages you might have loaded).

5.3.3 Challenge

Let's go ahead and download some core, important packages we will use for the rest of the course. Download (if you have not done so already) and load the following packages:

- `tidyverse`
- `rmarkdown`
- `knitr`
- `gapminder`
- `devtools`
- `stargazer`
- `rtweet`
- `kableExtra`

5.4 R Markdown

Throughout this course, we will be using R Markdown for lecture notes and homework assignments. R Markdown documents combine executable code, results, and prose commentary into one document. Think of an R Markdown files as a modern-day lab notebook, where you can capture not only what you did, but also what you were thinking.

The filename of an R Markdown document should end in `.Rmd` or `.rmd`. An R Markdown document can also be converted to an output format, like PDF, HTML, slideshows, Word files, and more.

R Markdown documents contain three important types of content:

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

5.4.1 YAML Header

YAML stands for "yet another markup language." R Markdown uses it to control many details of the output.

```
---
title: "Homework 1"
author: "Rochelle Terman"
date: "Fall 2019"
output: html_document
---
```

In this example, we specified the document's title, author, and date; we also specified that we want it to eventually be converted into an HTML document.

5.4.2 Markdown

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

````
Text formatting
------------------------------------------------------------

*italic*  or _italic_
**bold**   __bold__
`code`
superscript^2^ and subscript~2~

Headings
------------------------------------------------------------

# 1st Level Header

## 2nd Level Header
````

```
### 3rd Level Header

Lists
-------------------------------------------------------------

*   Bulleted list item 1

*   Item 2

    * Item 2a

    * Item 2b

1.  Numbered list item 1

1.  Item 2. The numbers are incremented automatically in the output.

Links and images
-------------------------------------------------------------

<http://example.com>

[linked phrase](http://example.com)

![optional caption text](path/to/img.png)

Tables
-------------------------------------------------------------

First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

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

5.4.3 Code Chunks

To run code inside an R Markdown document, you do it inside a "chunk." Think of a chunk like a step in a larger process. A chunk should be relatively self-contained and focused around a single task.

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

You can continue to run the code using the keyboard shortcut that we learned earlier: Cmd/Ctrl + Enter. You can also run the entire chunk by clicking the Run icon (it looks like a play button at the top of the chunk) or by pressing Cmd/Ctrl + Shift + Enter.

RStudio executes the code and displays the results inline with the code:

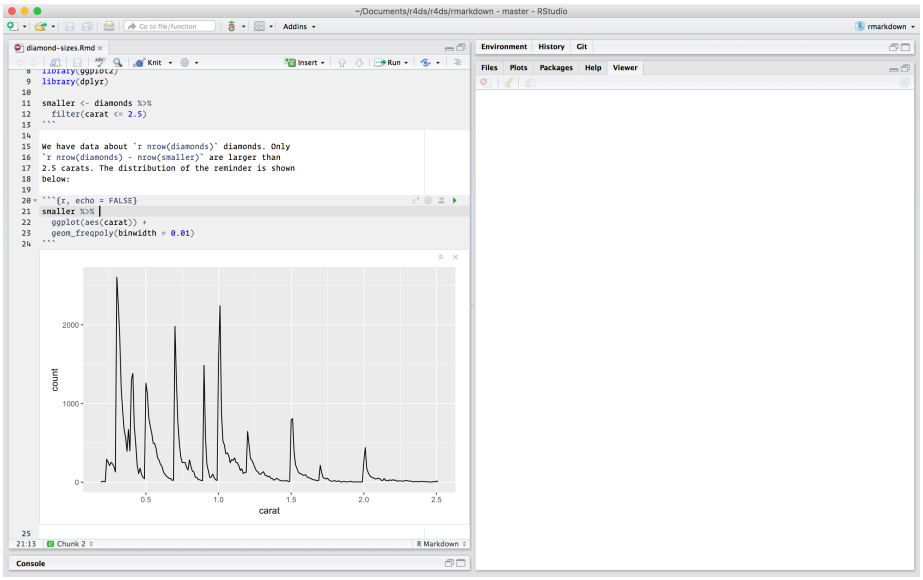


Figure 5.1:

5.4.4 Knitting

To produce a complete report containing all text, code, and results, click the "Knit" button at the top of the script editor (it looks like a ball of yarn) or press Cmd/Ctrl + Shift + K. This will display the report in the viewer pane and create a self-contained HTML file that you can share with others. The .html file is written in the same directory as your .Rmd file.

Knitting can be a finicky process that is sometimes challenging to troubleshoot. You will inevitably run into Knitting errors where RStudio will tell you that it is unable to knit your .Rmd file. When this happens, here are some approaches you can try out for troubleshooting:

1. **Read the error that RStudio gives you.** Usually, it will tell you which line in the code produced the error that stopped the Knitting process. Check out this line and see if there is a syntax error that needs to be fixed.
2. **Run every code chunk in order, one chunk at a time.** It is possible something will not run, which would cause the Knitting error. You can also try clearing your environment (in the top right pane) before running all the chunks.
3. **Have you copied and pasted text in from other sources?** Occasionally, an abnormal character copied from another app can cause a Knitting error.
4. **Check all of the file paths** and make sure they are accurate.

This list is by no means exhaustive. The most important step is step 1: read the error message. You can also try pasting it into Google to see how other R users have dealt with similar errors.

5.4.5 R Chunk Options for Knitting

You will notice that each R Chunk begins with `{r}`. Within these brackets, you can add "Chunk Options" to the R Chunk that will dictate how the R Chunk is treated when you Knit the `.Rmd`. Some commonly used options are:

- eval (default: TRUE): If FALSE, knitr will not run the code in the code chunk (it will, however, still display the code in the knitted document).
- include (default: TRUE): If FALSE, knitr will run the chunk but hide the code and its results in the final document.
- echo (default: TRUE): If FALSE, knitr will run the chunk and display the results but hide the code above its results in the final document.
- error (default: TRUE): If FALSE, knitr will not display any error messages generated by the code.
- message (default: TRUE): If FALSE, knitr will not display any messages generated by the code.
- warning (default: TRUE): If FALSE, knitr will not display any warnings generated by the code.

5.4.6 Cheatsheets and Other Resources

When working in RStudio, you can find an R Markdown cheatsheet by going to Help > Cheatsheets > R Markdown Cheat Sheet.

A helpful overview of R Markdown can also be found in R for Data Science.

A deep dive into R Markdown can be found here.

5.4.7 Challenges

Challenge 1.

Create a new R Markdown document with *File > New File > R Markdown…* Read the instructions. Practice running the chunks.

Now add some new markdown. Try adding some first-, second-, and third-level headers. Insert a link to a website.

Challenge 2.

In the first code chunk, modify `cars` to `mtcars`. Re-run the chunk and see modified output.

Challenge 3.

Knit the document into a PDF file. Verify that you can modify the input and see the output update.

Acknowledgments

This page is in part derived from the following sources:

1. R for Data Science, licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0.
2. Advanced R, licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License.
3. R Studio Support.
4. A ModernDive into R and the tidyverse.

Chapter 6

R Syntax

Frustration is natural when you start programming in R. R is a stickler for punctuation, and even one character out of place will cause it to complain. But while you should expect to be a little frustrated, take comfort in that it is both typical and temporary: it happens to everyone, and the only way to get over it is to keep trying.

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> **John Chambers**

6.1 Variables

6.1.1 Arithmetic

In its most basic form, R can be used as a simple calculator. Consider the following arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo (remainder): `%%`

```
1 / 200 * 30
#> [1] 0.15
```

```
(59 + 73 + 2) / 3
#> [1] 44.7
5 %% 2
#> [1] 1
```

But when we do this, none of our results are saved for later use.

6.1.2 Assigning Variables

An essential part of programming is creating objects (or variables).[1] Variables are names for values.

A variable is created when a value is assigned to it. We do that with `<-`.

```
x <- 3
```

`<-` is called the *assignment operator*. It assigns values on the right to objects on the left, like this:

```
object_name <- value
```

So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 goes into `x`.

NB: Do not use `=` for assignments. It will work in some contexts, but it will cause confusion later. There will be other scenarios where you will use `=` - we will discuss these later on.

We can use variables in calculations just as if they were values.

```
x <- 3
x + 5
#> [1] 8
```

Inspect objects to display values.

In R, the contents of an object can be printed by simply executing the object name.

```
x <- 3
x
#> [1] 3
```

[1] Technically, objects and variables are different things, but we will use the two interchangeably for now.

Whitespace makes code easier to read.

Notice that we surrounded `<-` with spaces. In `R`, white space is ignored (unlike Python). It is good practice to use spaces, because it makes code easier to read.

```
experiment<-"current vs. voltage"   # this is bad
experiment <- "current vs. voltage" # this is better
```

6.1.3 Variable Names

Object names can only contain letters, numbers, `_` and `.`.

You want your object names to be descriptive. `x` is not a good variable name (sorry!). You will also need a convention for multiple words. I recommend *snake_case*, where you separate lowercase words with `_`.

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

Let's make an assignment using snake_case:

```
r_rocks <- 2 ^ 3
```

And let's try to inspect it:

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

R is case-sensitive!

Use the TAB key to autocomplete.

Because typos are the absolute worst, we can use R Studio to help us type. Let's inspect `r_rocks` using RStudio's tab completion facility. Type "r_", press TAB, add characters until you have a unique prefix, then press return.

```
r_rocks
#> [1] 8
```

6.1.4 Challenges

Challenge 1: Making and Printing Variables

Make 3 variables: name (with your full name), city (where you were born), and year (when you were born).

Challenge 2: Swapping Values

Draw a table showing the values of the variables in this program after each statement is executed.

In simple terms, what do the last three lines of this program do?

```
lowest <- 1.0
highest <- 3.0
temp <- lowest
lowest <- highest
highest <- temp
```

Challenge 3: Predicting Values

What is the final value of `position` in the program below? Try to predict the value without running the program, then check your prediction.

```
initial <- "left"
position <- initial
initial <- "right"
```

Challenge 4: Syntax

Why does the following code fail?

```
age == 31
```

And the following?

```
31 <-  age
```

6.2 Functions

R has a large collection of built-in functions that helps us do things. When we use a function, we say we are *calling* a function.

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Here are some helpful built-in functions:

```
my_var <- c(1, 5, 2, 4, 5)

sum(my_var)
#> [1] 17
length(my_var)
#> [1] 5
min(my_var)
#> [1] 1
max(my_var)
#> [1] 5
unique(my_var)
#> [1] 1 5 2 4
```

6.2.1 Arguments

An *argument* is a value that is *passed* into a function. Every function returns a *result*.

Let's try using `seq()`, which makes regular sequences of numbers. While we are at it, we will learn more helpful features of RStudio.

Type `se` and hit TAB. A popup shows you possible completions. Specify `seq()` by typing more (a "q") to disambiguate, or by using ↑/↓ arrows to select. Notice the floating tooltip that pops up, reminding you of the function's arguments and purpose.

Press TAB once more when you have selected the function you want. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the arguments `1, 10` and hit return.

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

How many arguments did we pass into the `seq` function?

6.2.2 Store Function Output

Notice that, when we called the `seq` function, nothing changed in our environment. That is because we did not save our results to an object. Let's try it again by assigning a variable:

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

6.2.3 Argument Restrictions and Defaults

Let's use another function, called `round`:

```
round(60.123)
#> [1] 60
```

`round` must be given at least one argument. Moreover, it must be given things that can be meaningfully rounded.

```
round()
round('a')
```

Functions may have **default** values for some arguments.

By default, `round` will round off any number to zero decimal places. But we can specify the number of decimal places we want.

```
round(60.123)
#> [1] 60
round(60.123, digits = 2)
#> [1] 60.1
round(60.123, 2)
#> [1] 60.1
```

6.2.4 Documentation and Help Files

How do we know what kinds of arguments to pass into a function? Every built-in function comes with _**documentation**.

- `?` + object opens a help page for that specific object.
- `??` + object searches help pages containing the name of the object.

```
?mean
??mean
```

All help files are structured the same way:

- The **Arguments** section tells us exactly what kind of information we can pass into a function.
- The **Value** section explains what the output of the function is.
- The **Examples** section contains real examples of the function in use.

6.2.5 Challenges

Challenge 1: What Happens When

Explain, in simple terms, the order of operations in the following program: when does the addition happen, when does the subtraction happen, when is each function called, etc.

What is the final value of `radiance`?

```
radiance <- 1.0
radiance <- max(2.1, 2.0 + min(radiance, 1.1 * radiance - 0.5))
```

Challenge 2: Why?

Run the following code.

```
rich <- "gold"
poor <- "tin"
max(rich, poor)
```

Using the help files for `max`, explain why it returns the result it does.

6.3 Data Types

Every value in a program has a specific **type**. In R, those types are called "classes", and there are 4 of them:

- Character (text or "string").
- Numeric (integer or decimal).
- Integer (just integer).
- Logical (TRUE or FALSE booleans).

Example	Type
"a", "swc"	character
2, 15.5	numeric
2 (Must add a `L` at end to denote integer)	integer
`TRUE`, `FALSE`	logical

6.3.1 What Is that Type?

R is dynamically typed, meaning that it "guesses" what class a value is.

Use the built-in function `class()` to find out what type a value has.

```
class(3)
#> [1] "numeric"
class(3L)
#> [1] "integer"
class("Three")
#> [1] "character"
class(T)
#> [1] "logical"
```

This works on variables as well. But remember: the *value* has the type — the *variable* is just a label.

```
three <- 3
class(three)
#> [1] "numeric"

three <- "three"
class(three)
#> [1] "character"
```

A value's class determines what the program can do to it.

```
3 - 1
#> [1] 2
3 - "1"
#> Error in 3 - "1": non-numeric argument to binary operator
```

6.3.2 Coercion

We just learned we cannot subtract numbers and strings. Instead, use `as.` + name of class as a function to convert a value to a specified type.

```
3 - as.numeric("1")
```

This is called `coercion`. Here is another example:

```
my_var <- "FALSE"
my_var
#> [1] "FALSE"
as.logical(my_var)
#> [1] FALSE
```

What difference did you notice?

6.3.3 Other Objects

There are a few other "odd ball" types in R:

`NA` are missing values.

Missing values are specified with `NA`. `NA` will always be coerced to the correct type if used inside `c()`.

```
x <- c(NA, 1)
x
#> [1] NA  1
typeof(NA)
#> [1] "logical"
typeof(x)
#> [1] "double"
```

`Inf` is infinity.

You can have either positive or negative infinity.

```
1/0
#> [1] Inf
1/Inf
#> [1] 0
```

`NaN` means "not a number." It is an undefined value.

```
0/0
#> [1] NaN
```

6.3.4 Challenges

Challenge 1: Making and Coercing Variables

1. Make a variable `year` and assign it as the year you were born.
2. Coerce that variable to a string and assign it to a new variable `year_string`.
3. Someone in your class says they were born in 2001. Really? Really. Find out what your age difference is, using only `year_string`.

Challenge 2: Fix the Code

Change the following code to make the output `TRUE`:

```r
val_1 <- F
val_2 <- "F"

class(val_1) == class(val_2)
#> [1] FALSE
```

Chapter 7

Data Classes and Structures

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

This is **critical** to understand because these are the objects you will manipulate on a day-to-day basis in R. But they are not always as easy to work with as they seem at the outset. Dealing with object types and conversions is one of the most common sources of frustration for beginners.

R's base data structures can be organised by their dimensionality (1d, 2d, or nd) and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types). This gives rise to the five data types most often used in data analysis:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Dataframe
nd	Array	

Each data structure has its own specifications and behavior. In the rest of this chapter, we will cover the types of data objects that exist in R and their attributes.

1. Vectors
2. Lists
3. Factors
4. Matrices
5. Dataframes

7.1 Vectors

Let's start with one-dimensional (1d) objects. There are two kinds:

1. **Atomic vectors** - also called, simply, **vectors**.
2. **Lists**: Lists are distinct from atomic vectors because lists can contain other lists.

We'll discuss **atomic vectors** first.

7.1.1 Creating Vectors

Vectors are 1-dimensional chains of values. We call each value an *element* of a vector.

Atomic vectors are usually created with `c()`, which is short for 'combine':

```
x <- c(1, 2, 3)
x
#> [1] 1 2 3
length(x)
#> [1] 3
```

We can also add elements to the end of a vector by passing the original vector into the `c` function, like so:

```
z <- c("Beyonce", "Kelly", "Michelle", "LeToya")
z <- c(z, "Farrah")
z
#> [1] "Beyonce"  "Kelly"    "Michelle" "LeToya"   "Farrah"
```

Notice that vectors are always flat, even if you nest `c()`'s:

```
# these are equivalent
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

7.1.2 Naming a Vector

We can also attach names to our vector. This helps us understand what each element refers to.

You can give a name to the elements of a vector with the `names()` function. Have a look at this example:

```
days_month <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
names(days_month) <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov

days_month
#> Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
#>  31  28  31  30  31  30  31  31  30  31  30  31
```

You can name a vector when you create it:

```
some_vector <- c(name = "Rochelle Terman", profession = "Professor Extraordinaire")
some_vector
#>                         name                   profession
#>          "Rochelle Terman" "Professor Extraordinaire"
```

Notice that in the first case, we surrounded each name with quotation marks. But we don't have to do this when creating a named vector.

Names don't have to be unique, and not all values need to have a name associated. However, names are most useful for subsetting, described in the next chapter. When subsetting, it is most useful when the names are unique.

7.1.3 Calculations on Vectors

One of the most powerful things about vectors is that we can perform arithmetic calculations on them.

For example, we can sum up all the values in a numerical vector using **sum**:

```
a <- c(1, -2, 3)
sum(a)
#> [1] 2
```

We can also sum two vectors. It is important to know that if you **sum** two vectors in R, it takes the element-wise sum. For example, the following three statements are completely equivalent:

```
c(1, 2, 3) + c(4, 5, 6)
c(1 + 4, 2 + 5, 3 + 6)
c(5, 7, 9)
```

7.1.4 Types of Vectors

So there are there are four common types of vectors, depending on the class: * `logical` * `integer` * `numeric` (same as `double`) * `character`.

Logical Vectors

Logical vectors take on one of three possible values:

1. `TRUE`
2. `FALSE`
3. `NA` (missing value)

```
c(TRUE, TRUE, FALSE, NA)
#> [1]  TRUE  TRUE FALSE    NA
```

Numeric Vectors

Numeric vectors contain numbers. They can be stored as *integers* (whole numbers) or *doubles* (numbers with decimal points). In practice, you rarely need to concern yourself with this difference, but just know that they are different but related things.

```
c(1, 2, 335)
#> [1]   1   2 335
c(4.2, 4, 6, 53.2)
#> [1]  4.2  4.0  6.0 53.2
```

Character Vectors

Character vectors contain character (or 'string') values. Note that each value has to be surrounded by quotation marks *before* the comma.

```
c("Beyonce", "Kelly", "Michelle", "LeToya")
#> [1] "Beyonce"  "Kelly"    "Michelle" "LeToya"
```

7.1.5 Coercion

We can change or convert a vector's type using `as.....`

```
num_var <- c(1, 2.5, 4.5)
class(num_var)
#> [1] "numeric"
as.character(num_var)
#> [1] "1"   "2.5" "4.5"
```

Remember that all elements of a vector must be the same type. So when you attempt to combine different types, they will be **coerced** to the most "flexible" type.

For example, combining a character and an integer yields a character:

```
c("a", 1)
#> [1] "a" "1"
```

Guess what the following do without running them first:

```
c(1.7, "a")
c(TRUE, 2)
c("a", TRUE)
```

TRUE == 1 and FALSE == 0.

Notice that when a logical vector is coerced to an integer or double, `TRUE` becomes 1 and `FALSE` becomes 0. This is very useful in conjunction with `sum()` and `mean()`

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Total number of TRUEs
sum(x)
#> [1] 1

# Proportion that are TRUE
mean(x)
#> [1] 0.333
```

Coercion often happens automatically.

This is called *implicit coercion*. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any`, etc) will coerce to a logical. You will usually get a warning message if the coercion might lose information.

```
1 < "2"
#> [1] TRUE
"1" > 2
#> [1] FALSE
```

Sometimes coercions, especially nonsensical ones, won't work.

```
x <- c("a", "b", "c")
as.numeric(x)
#> Warning: NAs introduced by coercion
#> [1] NA NA NA
```

```
as.logical(x)
#> [1] NA NA NA
```

7.1.6 Challenges

Challenge 1: Create and examine your vector

Create a character vector called `fruit` that contain 4 of your favorite fruits. Then evaluate its structure using the commands below.

```
# First create your fruit vector
# YOUR CODE HERE


# Examine your vector
length(fruit)
class(fruit)
str(fruit)
```

Challenge 2: Coercion

```
# 1. Create a vector of a sequence of numbers between 1 to 10.

# 2. Coerce that vector into a character vector

# 3. Add the element "11" to the end of the vector

# 4. Coerce it back to a numeric vector.
```

Challenge 3: Calculations on Vectors

Create a vector of the numbers 11-20, and multiply it by the original vector from Challenge 2.

7.2 Lists

Lists are different from vectors because their elements can be of **any type**. Lists are sometimes called recursive vectors, because a list can contain other lists. This makes them fundamentally different from vectors.

7.2.1 Creating Lists

You construct lists by using `list()` instead of `c()`:

```
x <- list(1, "a", TRUE, c(4, 5, 6))
x
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]
#> [1] 4 5 6
```

7.2.2 Naming Lists

As with vectors, we can attach names to each element on our list:

```
my_list <- list(name1 = elem1,
                name2 = elem2)
```

This creates a list with components that are named `name1`, `name2`, and so on. If you want to name your lists after you've created them, you can use the `names()` function as you did with vectors. The following commands are fully equivalent to the assignment above:

```
my_list <- list(elem1, elem2)
names(my_list) <- c("name1", "name2")
```

7.2.3 List Structure

A very useful tool for working with lists is `str()` because it focuses on reviewing the structure of a list, not the contents.

```
x <- list(a = c(1, 2, 3),
          b = c("Hello", "there"),
          c = 1:10)
str(x)
#> List of 3
#>  $ a: num [1:3] 1 2 3
#>  $ b: chr [1:2] "Hello" "there"
#>  $ c: int [1:10] 1 2 3 4 5 6 7 8 9 10
```

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

```
x.vec <- c(1,2,3)
x.list <- list(1,2,3)
x.vec
#> [1] 1 2 3
x.list
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Lists are used to build up many of the more complicated data structures in R. For example, both data frames and linear models objects (as produced by `lm()`) are lists:

```
head(mtcars)
#>                    mpg cyl disp  hp drat   wt qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
#> Valiant           18.1   6  225 105 2.76 3.46 20.2  1  0    3    1
is.list(mtcars)
#> [1] TRUE
mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

You could say that a list is some kind super data type: you can store practically any piece of information in it!

For this reason, lists are extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

```
mod <- lm(mpg ~ wt, data = mtcars)
str(mod)
#> List of 12
#>  $ coefficients : Named num [1:2] 37.29 -5.34
#>   ..- attr(*, "names")= chr [1:2] "(Intercept)" "wt"
```

```
#>  $ residuals    : Named num [1:32] -2.28 -0.92 -2.09 1.3 -0.2 ...
#>   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" .
#>  $ effects      : Named num [1:32] -113.65 -29.116 -1.661 1.631 0.111 ...
#>   ..- attr(*, "names")= chr [1:32] "(Intercept)" "wt" "" "" ...
#>  $ rank         : int 2
#>  $ fitted.values: Named num [1:32] 23.3 21.9 24.9 20.1 18.9 ...
#>   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" .
#>  $ assign       : int [1:2] 0 1
#>  $ qr           :List of 5
#>   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
#>   .. ..- attr(*, "dimnames")=List of 2
#>   .. .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
#>   .. .. ..$ : chr [1:2] "(Intercept)" "wt"
#>   .. ..- attr(*, "assign")= int [1:2] 0 1
#>   ..$ qraux: num [1:2] 1.18 1.05
#>   ..$ pivot: int [1:2] 1 2
#>   ..$ tol  : num 1e-07
#>   ..$ rank : int 2
#>   ..- attr(*, "class")= chr "qr"
#>  $ df.residual  : int 30
#>  $ xlevels      : Named list()
#>  $ call         : language lm(formula = mpg ~ wt, data = mtcars)
#>  $ terms        :Classes 'terms', 'formula'  language mpg ~ wt
#>   .. ..- attr(*, "variables")= language list(mpg, wt)
#>   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
#>   .. .. ..- attr(*, "dimnames")=List of 2
#>   .. .. .. ..$ : chr [1:2] "mpg" "wt"
#>   .. .. .. ..$ : chr "wt"
#>   .. ..- attr(*, "term.labels")= chr "wt"
#>   .. ..- attr(*, "order")= int 1
#>   .. ..- attr(*, "intercept")= int 1
#>   .. ..- attr(*, "response")= int 1
#>   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
#>   .. ..- attr(*, "predvars")= language list(mpg, wt)
#>   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
#>   .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
#>  $ model        :'data.frame':   32 obs. of  2 variables:
#>   ..$ mpg: num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#>   ..$ wt : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
#>   ..- attr(*, "terms")=Classes 'terms', 'formula'  language mpg ~ wt
#>   .. .. ..- attr(*, "variables")= language list(mpg, wt)
#>   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
#>   .. .. .. ..- attr(*, "dimnames")=List of 2
#>   .. .. .. .. ..$ : chr [1:2] "mpg" "wt"
#>   .. .. .. .. ..$ : chr "wt"
```

```
#>   .. .. ..- attr(*, "term.labels")= chr "wt"
#>   .. .. ..- attr(*, "order")= int 1
#>   .. .. ..- attr(*, "intercept")= int 1
#>   .. .. ..- attr(*, "response")= int 1
#>   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
#>   .. .. ..- attr(*, "predvars")= language list(mpg, wt)
#>   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
#>   .. .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
#>  - attr(*, "class")= chr "lm"
```

7.2.4 Challenges

Challenge 1.

What are the four basic types of atomic vectors? How does a list differ from an atomic vector?

Challenge 2.

Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?

Challenge 3.

Create three vectors and combine them into a list. Assign them names.

Challenge 4.

If `x` is a list, what is the class of `x[1]`? How about `x[[1]]`?

7.3 Factors

Factors are special vectors that represent *categorical* data: variables that have a fixed and known set of possible values. Think: Democrat, Republican, Independent; Male, Female, Other; etc.

It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently.

Historically, factors were much easier to work with than characters. As a result, many of the functions in base R automatically convert characters to factors. This means that factors often pop up in places where they're not actually helpful.

7.3.1 Creating Factors

To create factors in R, you use the function `factor()`. The first thing that you have to do is create a vector that contains all the observations that belong to a limited number of categories. For example, `party_vector` contains the partyID of 5 different individuals:

```
party_vector <- c("Rep", "Rep", "Dem", "Rep", "Dem")
```

It is clear that there are two categories, or in R-terms **factor levels**, at work here: `Dem` and `Rep`.

The function `factor()` will encode the vector as a factor:

```
party_factor <- factor(party_vector)
party_vector
#> [1] "Rep" "Rep" "Dem" "Rep" "Dem"
party_factor
#> [1] Rep Rep Dem Rep Dem
#> Levels: Dem Rep
```

7.3.2 Summarizing a Factor

One of your favorite functions in R will be `summary()`. This will give you a quick overview of the contents of a variable. Let's compare using `summary()` on both the character vector and the factor:

```
summary(party_vector)
#>    Length     Class      Mode
#>         5 character character
summary(party_factor)
#> Dem Rep
#>   2   3
```

7.3.3 Changing Factor Levels

When you create the factor, the factor levels are set to specific values. We can access those values with the `levels()` function.

```
levels(party_factor)
#> [1] "Dem" "Rep"
```

Any values *not* in the set of levels will be silently converted to `NA`. Let's say we want to add an Independent to our sample:

```
party_factor[5] <- "Ind"
#> Warning in `[<-.factor`(`*tmp*`, 5, value = "Ind"): invalid factor level,
```

```
#> NA generated
party_factor
#> [1] Rep  Rep  Dem  Rep  <NA>
#> Levels: Dem Rep
```

We first need to add "Ind" to our factor levels. This will allow us to add Independents to our sample:

```
levels(party_factor)
#> [1] "Dem" "Rep"
levels(party_factor) <- c("Dem", "Rep", "Ind")

party_factor[5] <- "Ind"
party_factor
#> [1] Rep Rep Dem Rep Ind
#> Levels: Dem Rep Ind
```

7.3.4 Factors are Integers

Factors are pretty much integers that have labels on them. Underneath, it's really numbers (1, 2, 3...).

```
str(party_factor)
#>  Factor w/ 3 levels "Dem","Rep","Ind": 2 2 1 2 3
```

They are better than using simple integer labels because factors are self-describing. For example, `democrat` and `republican` are more descriptive than `1`s and `2`s.

However, **factors are NOT characters!!**

While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings.

```
x <- c("a", "b", "b", "a")
x <- as.factor(x)
c(x, "c")
#> [1] "1" "2" "2" "1" "c"
```

For this reason, it's usually best to explicitly **convert** factors to character vectors if you need string-like behaviour.

```
x <- c("a", "b", "b", "a")
x <- as.factor(x)
x <- as.character(x)
c(x, "c'")
#> [1] "a"  "b"  "b"  "a"  "c'"
```

7.3.5 Challenges

Challenge 1.

What happens to a factor when you modify its levels?

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
f1
#>  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
#> Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

Challenge 2.

What does this code do? How do `f2` and `f3` differ from `f1`?

```
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

7.4 Matrices

Matrices are 2-d vectors. That is, they are a collection of elements of the same data type (numeric, character, or logical), arranged into a fixed number of rows and columns.

By definition, if you want to combine different types of data (one column numbers, another column characters), you want a **dataframe**, not a matrix.

7.4.1 Creating Matrices

We can create a matrix using the `matrix()` function. In this function, we assign dimensions to a vector, like this:

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

Notice that matrices fill column-wise. We can change this using the `byrow` argument:

```
m <- matrix(1:6, byrow = T, nrow = 2, ncol = 3)
m
#>      [,1] [,2] [,3]
```

```
#> [1,]    1    2    3
#> [2,]    4    5    6
```

Another way to create matrices 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
# or
rbind(x, y)
#>   [,1] [,2] [,3]
#> x    1    2    3
#> y   10   11   12
```

7.4.2 Matrix Dimensions

Use `dim()` to find out how many rows or columns are in a matrix (or dataframe)

```
dim(m)
#> [1] 2 3
```

We can transpose a matrix (or dataframe) with `t()`

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
t(m)
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
#> [3,]    5    6
```

7.4.3 Matrix Names

Just like vectors or lists, we can give matrices names that describe the rows and columns

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

rownames(m) <- c("row1", "row2")
colnames(m) <- c("A", "B", "C")

m
#>      A B C
#> row1 1 3 5
#> row2 2 4 6
```

7.4.4 Challenge

Take a look at the vector I've created about box office sales for the first three Harry Potter movies:

```
# Box office sales (in millions!)
philosophers_stone <- c(66.1, 317.6, 657.2)
chamber_secrets <- c(54.7, 261.9, 616.9)
prisoner_azkaban <- c(45.6, 249.5, 547.1)

# Vectors region and titles, used for naming
region <- c("UK", "US", "Other")
titles <- c("Philosopher's Stone", "Chamber of Secrets", "Prisoner of Azkaban")
```

Your challenge is to:

1. Combine the first three vectors into a matrix
2. Add names for the matrix's rows (`titles`) and columns (`region`)
3. Use `rowSums()` to find the total Worldwide Box Office sales for each movie.

7.5 Dataframes

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

Let's say we're working with the following survey data:

- 'Are you married?' or 'yes/no' questions (`logical`)
- 'How old are you?' (`numeric`)
- 'What is your opinion on Trump?' or other 'open-ended' questions (`character`)
- ...

A matrix won't work here because the dataset contains different data types.

A dataframe is a 2-dimentional data structure containing heterogeneous data types. Each column is a variable of a dataset, and the rows are observations.

> NB: You might have heard of "tibbles," used in the `tidyverse` suite of packages. Tibbles are like dataframes 2.0, tweaking some of the behavior of dataframes to make life easier for data anlysis. For now, just think of tibbles and dataframes as the same thing and don't worry about the difference.

7.5.1 Creating Dataframes

R contains a number of built-in datasets that are stored as dataframes. For example, the `mtcars` dataset contains information on automobile design and performance for 32 automobiles:

```
class(mtcars)
#> [1] "data.frame"
head(mtcars)
#>                   mpg cyl disp  hp drat   wt qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
#> Valiant           18.1   6  225 105 2.76 3.46 20.2  1  0    3    1
```

We also create dataframes when we import data through `read.csv` or other data file input. We'll talk more about importing data later in the class.

We can create a dataframe from scratch using `data.frame()`. This function takes vectors as input:

```
# Definition of vectors
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet", "Terrestria
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

planets <- data.frame(name, type, diameter, rings)
planets
#>      name               type diameter rings
#> 1 Mercury Terrestrial planet    0.382 FALSE
#> 2   Venus Terrestrial planet    0.949 FALSE
#> 3   Earth Terrestrial planet    1.000 FALSE
#> 4    Mars Terrestrial planet    0.532 FALSE
#> 5 Jupiter          Gas giant   11.209  TRUE
#> 6  Saturn          Gas giant    9.449  TRUE
```

```
#> 7  Uranus          Gas giant    4.007  TRUE
#> 8 Neptune          Gas giant    3.883  TRUE
```

Beware: `data.frame()`'s default behaviour turns strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour as needed:

```
planets <- data.frame(name, type, diameter, rings, stringsAsFactors = F)
planets
#>      name               type diameter rings
#> 1 Mercury Terrestrial planet    0.382 FALSE
#> 2   Venus Terrestrial planet    0.949 FALSE
#> 3   Earth Terrestrial planet    1.000 FALSE
#> 4    Mars Terrestrial planet    0.532 FALSE
#> 5 Jupiter          Gas giant   11.209  TRUE
#> 6  Saturn          Gas giant    9.449  TRUE
#> 7  Uranus          Gas giant    4.007  TRUE
#> 8 Neptune          Gas giant    3.883  TRUE
```

7.5.2 The Structure of Dataframes

Under the hood, a dataframe is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list.

```
vec1 <- 1:3
vec2 <- c("a", "b", "c")
df <- data.frame(vec1, vec2)

str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ vec1: int  1 2 3
#>  $ vec2: Factor w/ 3 levels "a","b","c": 1 2 3
```

The `length()` of a dataframe is the length of the underlying list and so is the same as `ncol()`; `nrow()` gives the number of rows.

```
vec1 <- 1:3
vec2 <- c("a", "b", "c")
df <- data.frame(vec1, vec2)

# these two are equivalent - number of columns
length(df)
#> [1] 2
ncol(df)
#> [1] 2

# get number of rows
```

```
nrow(df)
#> [1] 3

# get number of both columns and rows
dim(df)
#> [1] 3 2
```

7.5.3 Naming Dataframes

Like matrices, dataframes have `colnames()`, and `rownames()`. However, since dataframes are really lists (of vectors) under the hood `names()` and `colnames()` are the same thing.

```
vec1 <- 1:3
vec2 <- c("a", "b", "c")
df <- data.frame(vec1, vec2)

# these two are equivalent
names(df)
#> [1] "vec1" "vec2"
colnames(df)
#> [1] "vec1" "vec2"

# change the colnames
colnames(df) <- c("Number", "Character")
df
#>   Number Character
#> 1      1         a
#> 2      2         b
#> 3      3         c

names(df) <- c("Number", "Character")
df
#>   Number Character
#> 1      1         a
#> 2      2         b
#> 3      3         c

# change the rownames
rownames(df)
#> [1] "1" "2" "3"
rownames(df) <- c("donut", "pickle", "pretzel")
df
#>         Number Character
#> donut        1         a
```

```
#> pickle       2          b
#> pretzel      3          c
```

7.5.4 Coercing Dataframes

Coerce an object to a dataframe with `as.data.frame()`:

- A vector will create a one-column dataframe.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

7.5.5 Challenges

Challenge 1.

Create a 3x2 data frame called `basket`. The first column should contain the names of 3 fruits. The second column should contain the price of those fruits.

Challenge 2.

Now give your dataframe appropriate column and row names.

Challenge 3.

Add a third column called `color`, that tells me what color each fruit is.

7.5.6 Quiz

You can check your answers in answers.

1. How is a list different from an vector?
2. What are the four common types of vectors?
3. What are names? How do you get them and set them?
4. How is a matrix different from a data frame?

7.5.7 Answers

1. The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type.
2. The four common types of vector are logical, integer, double (sometimes called numeric), and character.
3. Names allow you to attach labels to values. You can get and set individual names with `names(x)` and `names(x) <- c("x", "y", ...)`.
4. Every element of a matrix must be the same type; in a data frame, the different columns can have different types.

Chapter 8

Subsetting

When working with data, you'll need to subset objects early and often. Luckily, R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match. Subsetting is hard to learn because you need to master a number of interrelated concepts:

- The three subsetting operators: `[`, `[[`, and `$`.
- The four types of subsetting.
- The important differences in behaviour for different objects (e.g., vectors, lists, factors, matrices, and data frames).
- The use of subsetting in conjunction with assignment.

This unit helps you master subsetting by starting with the simplest type of subsetting: subsetting an atomic vector with `[`. It then gradually extends your knowledge, first to more complicated data types (like dataframes and lists), and then to the other subsetting operators, `[[` and `$`. You'll then learn how subsetting and assignment can be combined to modify parts of an object, and, finally, you'll see a large number of useful applications.

8.1 Subsetting Vectors

It's easiest to learn how subsetting works for vectors, and then how it generalises to higher dimensions and other more complicated objects. We'll start with `[`, the most commonly used operator.

8.1.1 Subsetting Types

Let's explore the different types of subsetting with a simple vector, `x`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Note that the number after the decimal point gives the original position in the vector.

There are four things you can use to subset a vector:

1. Positive integers return elements at the specified positions:

```
(x <- c(2.1, 4.2, 3.3, 5.4))
#> [1] 2.1 4.2 3.3 5.4
x[1]
#> [1] 2.1
```

We can also index multiple values by passing a vector of integers:

```
(x <- c(2.1, 4.2, 3.3, 5.4))
#> [1] 2.1 4.2 3.3 5.4
x[c(3, 1)]
#> [1] 3.3 2.1

# Duplicated indices yield duplicated values
x[c(1, 1)]
#> [1] 2.1 2.1
```

Note that you *have* to use `c` inside the `[` for this to work!

More examples:

```
# `order(x)` gives the index positions of smallest to largest values.
(x <- c(2.1, 4.2, 3.3, 5.4))
#> [1] 2.1 4.2 3.3 5.4
order(x)
#> [1] 1 3 2 4

# use this to order values.
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4
x[c(1, 3, 2, 4)]
#> [1] 2.1 3.3 4.2 5.4
```

2. Negative integers omit elements at the specified positions:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[-1]
#> [1] 4.2 3.3 5.4
x[-c(3, 1)]
#> [1] 4.2 5.4
```

You can't mix positive and negative integers in a single subset:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[c(-1, 2)]
#> Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

3. Character vectors return elements with matching names. This only works if the vector is named.

```
x <- c(2.1, 4.2, 3.3, 5.4)

# apply names
names(x) <- c("a", "b", "c", "d")

# subset using names
x[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Like integer indices, you can repeat indices
x[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# Careful! Names are always matched exactly
x <- c(abc = 1, def = 2)
x[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

4. Logical vectors select elements where the corresponding logical value is `TRUE`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[c(TRUE, TRUE, FALSE, FALSE)]
```

```
#> [1] 2.1 4.2
```

8.1.2 Conditional Subsetting

Logical subsetting is the most useful type of subsetting, because you use it to subset based on **conditional** or **comparative** statements.

The (logical) comparison operators known to R are:

- `<` for less than
- `>` for greater than
- `<=` for less than or equal to
- `>=` for greater than or equal to
- `==` for equal to each other
- `!=` not equal to each other

The nice thing about R is that you can use these comparison operators also on vectors. For example:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x > 3
#> [1] FALSE  TRUE  TRUE  TRUE
```

This command tests for every element of the vector if the condition stated by the comparison operator is `TRUE` or `FALSE`. And it returns a logical vector!

We can now pass this statement between the square brackets that follow `x` to subset only those items that match `TRUE`:

```
x[x > 3]
#> [1] 4.2 3.3 5.4
```

You can combine conditional statements with `&` (and), `|` (or), and `!` (not)

```
x <- c(2.1, 4.2, 3.3, 5.4)

# combing two conditional statements with &
x > 3 & x < 5
#> [1] FALSE  TRUE  TRUE FALSE
x[x > 3 & x < 5]
#> [1] 4.2 3.3

# combing two conditional statements with |
x < 3 | x > 5
#> [1]  TRUE FALSE FALSE  TRUE
x[x < 3 | x > 5]
#> [1] 2.1 5.4
```

```
# combining conditional statements with !
!x > 5
#> [1]  TRUE  TRUE  TRUE FALSE
x[!x > 5]
#> [1] 2.1 4.2 3.3
```

Another way to generate implicit conditional statements is using the `%in%` operator, which tests whether an item is in a set:

```
x <- c(2.1, 4.2, 3.3, 5.4)

# generate implicit logical vectors through the %in% operator
x %in% c(3.3, 4.2)
#> [1] FALSE  TRUE  TRUE FALSE
x[x %in% c(3.3, 4.2)]
#> [1] 4.2 3.3
```

8.1.3 Challenge

Subset `country.vector` below to return every value EXCEPT "Canada" and "Brazil"

```
country.vector<-c("Afghanistan", "Canada", "Sierra Leone", "Denmark", "Japan", "Brazil")

# Do it using positive integers

# Do it using negative integers

# Do it using a logical vector

# Do it using a conditional statement (and an implicit logical vector)
```

8.2 Subsetting Lists

Subsetting a list works in the same way as subsetting an atomic vector. However, there's one important difference: `[` will always return a list. `[[` and `$`, as described below, let you pull out the components of the list.

Let's illustrate with the following list `my_list`:

```
my_list <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

8.2.1 With [

[extracts a sub-list where the result will always be a list. Like with vectors, you can subset with a logical, integer, or character vector.

```
my_list[1:2]
#> $a
#> [1] 1 2 3
#>
#> $b
#> [1] "a string"
str(my_list[1:2])
#> List of 2
#>  $ a: int [1:3] 1 2 3
#>  $ b: chr "a string"

my_list[4]
#> $d
#> $d[[1]]
#> [1] -1
#>
#> $d[[2]]
#> [1] -5
str(my_list[4])
#> List of 1
#>  $ d:List of 2
#>   ..$ : num -1
#>   ..$ : num -5

my_list["a"]
#> $a
#> [1] 1 2 3
str(my_list["a"])
#> List of 1
#>  $ a: int [1:3] 1 2 3
```

8.2.2 With [[

[[extracts a single *component* from a list. In other words, it removes that hierarchy and returns whatever object is stored inside.

```
my_list[[1]]
#> [1] 1 2 3
str(my_list[[1]])
#>  int [1:3] 1 2 3
```

```
# compare to
my_list[1]
#> $a
#> [1] 1 2 3
str(my_list[1])
#> List of 1
#>  $ a: int [1:3] 1 2 3
```

The distinction between `[` and `[[` is really important for lists, because `[[` drills down into the list while `[` returns a new, smaller list.

> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6."
>
> — **?**

8.2.3 with $

`$` is a shorthand for extracting named elements of a list. It works similarly to `[[` except that you don't need to use quotes.

```
my_list$a
#> [1] 1 2 3

# same as
my_list[["a"]]
#> [1] 1 2 3
```

The `$` operator becomes especially helpful when applied to dataframes, explained more below.

8.2.4 Challenge

Take a look at the linear model below:

```
mod <- lm(mpg ~ wt, data = mtcars)
summary(mod)
#>
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -4.543 -2.365 -0.125  1.410  6.873
#>
#> Coefficients:
```

```
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   37.285      1.878   19.86  < 2e-16 ***
#> wt            -5.344      0.559   -9.56  1.3e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.05 on 30 degrees of freedom
#> Multiple R-squared:  0.753,  Adjusted R-squared:  0.745
#> F-statistic: 91.4 on 1 and 30 DF,  p-value: 1.29e-10
```

Extract the R squared from the model summary.

8.3 Subsetting Matrices

Similar to vectors, you can use the square brackets `[ ]` to select one or multiple elements from a matrix. But whereas vectors have one dimension, matrices have two dimensions. We therefore have to use two subsetting vectors – one for rows to select, another for columns – separated by a comma.

Check out the following matrix:

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
#> [3,] 3 6 9
```

We can subset this matrix by passing two subsetting vectors: one to select rows, another to select columns:

```
# selects the value at the first row and second column
a[1, 2]
#> B
#> 4

# selects first row, and the first and third columns
a[1, -2]
#> A C
#> 1 7

# selects first two rows, and the first and third columns
a[c(1,2), c(1, 3)]
#>      A C
```

```
#> [1,] 1 7
#> [2,] 2 8
```

Blank subsetting is also useful because it lets you keep all rows or all columns.

```
a[c(1, 2), ] # selects first two rows and all columns
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
```

8.4 Subsetting Dataframes

Data from data frames can be addressed like matrices, using two vectors separated by a comma.

```
# Definition of vectors
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

planets <- data.frame(name, type, diameter, rings, stringsAsFactors = F)
planets
#>      name               type diameter rings
#> 1 Mercury Terrestrial planet    0.382 FALSE
#> 2   Venus Terrestrial planet    0.949 FALSE
#> 3   Earth Terrestrial planet    1.000 FALSE
#> 4    Mars Terrestrial planet    0.532 FALSE
#> 5 Jupiter          Gas giant   11.209  TRUE
#> 6  Saturn          Gas giant    9.449  TRUE
#> 7  Uranus          Gas giant    4.007  TRUE
#> 8 Neptune          Gas giant    3.883  TRUE
```

Let's try some subsetting now.

```
# Print out diameter of Mercury (row 1, column 3)
planets[1, 3]
#> [1] 0.382

# Print out data for Mars (entire fourth row)
planets[4, ]
#>   name               type diameter rings
#> 4 Mars Terrestrial planet    0.532 FALSE

# Print first two rows of the first two columns
```

```
planets[1:2, 1:2]
#>      name               type
#> 1 Mercury Terrestrial planet
#> 2   Venus Terrestrial planet
```

8.4.1 Subsetting Names and $

Instead of using numerics to select elements of a data frame, you can also use the variable names to select columns of a data frame.

Suppose you want to select the first three elements of the type column. One way to do this is

```
planets[1:3, 2]
#> [1] "Terrestrial planet" "Terrestrial planet" "Terrestrial planet"
```

A possible disadvantage of this approach is that you have to know (or look up) the column number of type, which gets hard if you have a lot of variables. It is often easier to just make use of the variable name:

```
planets[1:3, "type"]
#> [1] "Terrestrial planet" "Terrestrial planet" "Terrestrial planet"
```

You will often want to select an entire column, namely one specific variable from a data frame. If you want to select all elements of the variable "diameter", for example, both of these will do the trick:

```
planets[,3]
#> [1]  0.382  0.949  1.000  0.532 11.209  9.449  4.007  3.883
planets[,"diameter"]
#> [1]  0.382  0.949  1.000  0.532 11.209  9.449  4.007  3.883
```

However, there is a short-cut. If your columns have names, you can use the `$` sign:

```
planets$diameter
#> [1]  0.382  0.949  1.000  0.532 11.209  9.449  4.007  3.883
```

Remember that datasets are really lists of vectors (one vector per column). Just as `list$name` selects the `name` element from the list, `df$name` selects the `name` column (vector) from the dataframe.

8.4.2 Conditional Subsetting

What if we want to subset the dataset based on some condition? Let's say we want to extract all the planets with a diameter greater than 3? We could inspect

the dataset and record all the observations that fit that description, but that's tedious and error prone.

There's a better way! We can combine two powerful subsetting tools: the `$` operator and conditional subsetting.

First, we extract the `diameter` column.

```
diameters <- planets$diameter
```

Then, we find the elements that are greater than 3.

```
diameters > 3
#> [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

It's a boolean vector! We can now use this inside `[ , ]` to extract all plantes with `diameter > 3`.

Think: Are we subsettings row or columns here?

```
planets[diameters > 3, ]
#>      name      type diameter rings
#> 5 Jupiter Gas giant    11.21  TRUE
#> 6  Saturn Gas giant     9.45  TRUE
#> 7  Uranus Gas giant     4.01  TRUE
#> 8 Neptune Gas giant     3.88  TRUE

# same as
# planets[planets$diameter > 3, ]
```

Because it allows you to easily combine conditions from multiple columns, logical subsetting is probably the most commonly used technique for extracting rows out of a data frame.

8.4.3 List-Like and Matrix-Like Subsetting

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists, and return only the columns.

```
df <- data.frame(x = 4:6, y = 3:1, z = letters[1:3])

# Like a list:
df[c("x", "z")]
#>   x z
#> 1 4 a
#> 2 5 b
#> 3 6 c

# Like a matrix
```

```
df[, c("x", "z")]
#>   x z
#> 1 4 a
#> 2 5 b
#> 3 6 c
```

But there's an important difference when you select a single column: matrix subsetting simplifies by default, list subsetting does not.

```
df <- data.frame(x = 4:6, y = 3:1, z = letters[1:3])

# like a list
df["x"]
#>   x
#> 1 4
#> 2 5
#> 3 6
class(df["x"])
#> [1] "data.frame"

# like a matrix
df[, "x"]
#> [1] 4 5 6
class(df[, "x"])
#> [1] "integer"
```

8.4.4 Challenges

Challenge 1.

Fix each of the following common data frame subsetting errors:

```
# check out what we're dealing with
mtcars

# fix
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

Challenge 2.

Why does `mtcars[1:20]` return an error? How does it differ from the similar `mtcars[1:20, ]`?

8.5 Sub-assignment

8.5.1 Basics of Sub-assignment

All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5
```

This is especially useful when conditionally modifying vectors. For example, let's say we wanted to replace all values less than 3 with NA.

```
x <- 1:5
x[x < 3] <- NA
x
#> [1] NA NA  3  4  5
```

This also works on dataframes. Let's say we wanted to modify our `planets` dataframe.

```
# Definition of vectors
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

planets <- data.frame(name, type, diameter, rings, stringsAsFactors = F)
planets
#>      name               type diameter rings
#> 1 Mercury Terrestrial planet    0.382 FALSE
#> 2   Venus Terrestrial planet    0.949 FALSE
#> 3   Earth Terrestrial planet    1.000 FALSE
#> 4    Mars Terrestrial planet    0.532 FALSE
#> 5 Jupiter          Gas giant   11.209  TRUE
#> 6  Saturn          Gas giant    9.449  TRUE
#> 7  Uranus          Gas giant    4.007  TRUE
#> 8 Neptune          Gas giant    3.883  TRUE
```

Let's say we want to replace the term "Terrestrial planet" with "TP". First we need to subset `type` for those elements:

```
planets$type == "Terrestrial planet"
#> [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Now we can re-assign the values of `type`:

```
planets$type[planets$type == "Terrestrial planet"]
#> [1] "Terrestrial planet" "Terrestrial planet" "Terrestrial planet"
#> [4] "Terrestrial planet"
planets$type[planets$type == "Terrestrial planet"] <- "TP"
planets
#>      name      type diameter rings
#> 1 Mercury        TP    0.382 FALSE
#> 2   Venus        TP    0.949 FALSE
#> 3   Earth        TP    1.000 FALSE
#> 4    Mars        TP    0.532 FALSE
#> 5 Jupiter Gas giant   11.209  TRUE
#> 6  Saturn Gas giant    9.449  TRUE
#> 7  Uranus Gas giant    4.007  TRUE
#> 8 Neptune Gas giant    3.883  TRUE
```

8.5.2 Recycling

When applying an operation to two vectors that requires them to be the same length, R automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one.

```
df <- data.frame(x = 4:7, y = letters[1:4])

# r recycles values
df$x <- c(1, 2)
df
#>   x y
#> 1 1 a
#> 2 2 b
#> 3 1 c
#> 4 2 d

# sometimes this is helpful if you want to replace an entire vector to one value.
df$x <- df$x + 3
df
#>   x y
#> 1 4 a
#> 2 5 b
```

```
#> 3 4 c
#> 4 5 d
```

8.5.3 Applications

The basic principles described above give rise to a wide variety of useful applications. Some of the most important applications are described below. Many of these basic techniques are wrapped up into more concise functions (e.g., `subset()`, `merge()`, `plyr::arrange()`), but it is useful to understand how they are implemented with basic subsetting. This will allow you to adapt to new situations that are not dealt with by existing functions.

Ordering Columns

Consider we have this data frame:

```
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)
df
#>          Country         Region Language
#> 1           Iraq    Middle East   Arabic
#> 2          China           Asia Mandarin
#> 3         Mexico  North America  Spanish
#> 4         Russia Eastern Europe  Russian
#> 5 United Kingdom Western Europe  English
```

What if we wanted to reorder the columns so that `Region` is first? We can do so using subsetting with the names (or number) of the columns:

```
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

# reorder columns using names
names(df)
#> [1] "Country"  "Region"   "Language"
df1 <- df[, c("Region", "Country", "Language")]
df1
#>           Region        Country Language
```

```
#> 1    Middle East            Iraq   Arabic
#> 2           Asia           China Mandarin
#> 3  North America          Mexico  Spanish
#> 4 Eastern Europe          Russia  Russian
#> 5 Western Europe United Kingdom  English

# reorder columns using indices
names(df)
#> [1] "Country"  "Region"   "Language"
df1 <- df[, c(2,1,3)]
df1
#>           Region        Country Language
#> 1    Middle East           Iraq   Arabic
#> 2           Asia          China Mandarin
#> 3  North America          Mexico  Spanish
#> 4 Eastern Europe          Russia  Russian
#> 5 Western Europe United Kingdom  English
```

One helpul function is the `order` function. It takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered:

```
x <- c("b", "c", "a")
order(x)
#> [1] 3 1 2
x[order(x)]
#> [1] "a" "b" "c"
```

Knowing this, we can use `order` to reorder our columns by alphabetical order.

Removing (or keeping) columns from data frames.

There are two ways to remove columns from a data frame. You can set individual columns to `NULL`:

```
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

df$Language <- NULL
```

Or you can subset to return only the columns you want:

```
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe
```

```
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

df1 <- df[, c("Country", "Region")]
df1
#>           Country         Region
#> 1            Iraq    Middle East
#> 2           China           Asia
#> 3          Mexico  North America
#> 4          Russia Eastern Europe
#> 5 United Kingdom Western Europe

# using negative integers
df2 <- df[, -3]
df2
#>           Country         Region
#> 1            Iraq    Middle East
#> 2           China           Asia
#> 3          Mexico  North America
#> 4          Russia Eastern Europe
#> 5 United Kingdom Western Europe
```

Chapter 9

Project Workflow

One day...

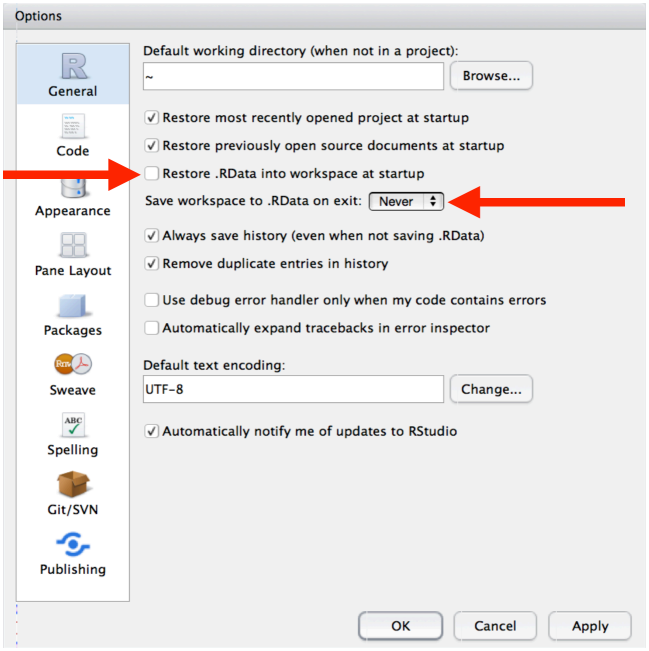
- you will need to quit R, go do something else, and return to your analysis the next day.
- you will be working on multiple projects simultaneously, and you'll want to keep them separate.
- you will need to bring data from the outside world into R, and send numerical results and figures from R back out into the world.

This unit will teach you how to set up your workflow to make the best use of R.

9.1 Organizing Code

9.1.1 Store Analyses in Scripts, Not Workspaces.

R Studio automatically preserves your workspace (environment and command history) when you quit R, and re-loads it the next session. I recommend you turn this behavior off.



This will cause you some short-term pain, because now when you restart RStudio, it will not remember the results of the code that you ran last time. But this short-term pain will save you long-term agony, because it forces you to capture all important interactions in your scripts.

9.1.2 Working Directories and Paths

Like many programming languages, R has a powerful notion of the **working directory**. This is where R looks for files that you ask it to load, and where it will put any files that you ask it to save.

RStudio shows your current working directory at the top of the console. You can print this out in R code by running `getwd()`:

```
getwd()
#> [1] "/Users/rochelleterman/Desktop/course-site"
```

I do not recommend it, but you can set the working directory from within R:

```
setwd("/path/to/my/CoolProject")
```

The command above prints out a **path** to your working directory. Think of a path as an address. Paths are incredibly important in programming, but can be a little tricky. Let's go into a bit more detail.

Absolute Paths

Absolute paths are paths that point to the same place regardless of your current working directory. They always start with the **root directory** that holds everything else on your computer.

- In Windows, absolute paths start with a drive letter (e.g. `C:`) or two backslashes (e.g. `\\servername`).
- In Mac/Linux they start with a slash `/`. This is the leading slash in `/users/rterman`.

Inside the root directory are several other directories, which we call **subdirectories**. We know that the directory `/home/rterman` is stored inside `/home` because `/home` is the first part of its name. Similarly, we know that `/home` is stored inside the root directory `/` because its name begins with `/`.

> Notice that there are two meanings for the `/` character. When it appears at the front of a file or directory name, > it refers to the root directory. When it appears *inside* a name, > it's just a separator.

Mac/Linux vs. Windows

There are two basic styles of paths: Mac/Linux and Windows. The main difference is how you separate the components of the path. Mac and Linux use slashes (e.g. `plots/diamonds.pdf`) and Windows uses backslashes (e.g. `plots\diamonds.pdf`).

R can work with either type, no matter what platform you're currently using. Unfortunately, backslashes mean something special to R, and to get a single backslash in the path, you need to type two backslashes! That makes life frustrating, so I recommend always using the Linux/Mac style with forward slashes.

Home Directory

Sometimes you'll see a `~` character in a path.

- In Mac/Linux, the `~` is a convenient shortcut to your **home directory** (`/users/rterman`).
- Windows doesn't really have the notion of a home directory, so it usually points to your documents directory (`C:\Documents and Settings\rterman`)

Absolute vs. Relative Paths

You should try not to use absolute paths in your scripts, because they hinder sharing: no one else will have exactly the same directory configuration as you.

Another way to direct R to something is to give it a **relative path**.

Relative paths point to something relative to where you are (i.e. relative to your working directory), rather than from the root of the file system. For example, if your current working directory is `/home/rterman`, then the relative path `data/un.csv` directs to the full absolute path: `/home/rterman/data/un.csv`.

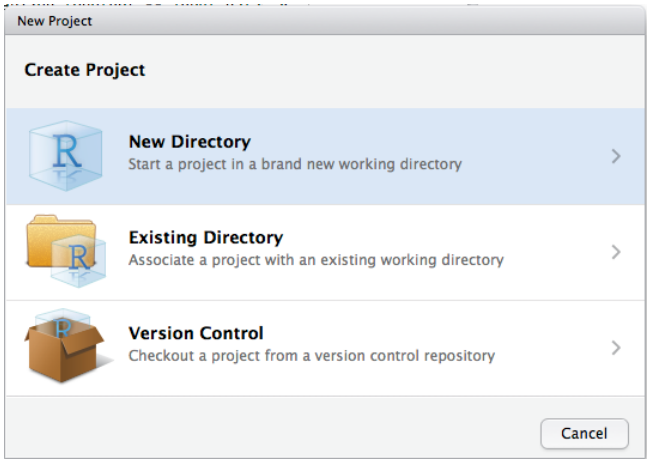
9.1.3 R Projects

As a beginning R user, it's OK to let your home directory, documents directory, or any other weird directory on your computer be R's working directory.

But from this point forward, you should be organizing your projects into dedicated subdirectories, containing all the files associated with a project — input data, R scripts, results, figures.

This is such a common practice that RStudio has built-in support for this via **projects**.

Let's make a project together. Click `File > New Project`, then:



Think carefully about which subdirectory you put the project in. If you don't store it somewhere sensible, it will be hard to find in the future!

Once this process is complete, you'll get a new RStudio project. Check that the "home" directory of your project is the current working directory:

```
getwd()
#> [1] "/Users/rochelleterman/Desktop/course-site"
```

Now whenever you refer to a file with a relative path, it will look for it there.

Go ahead and create a new R script and save it inside the project folder.

Quit RStudio. Inspect the folder associated with your project — notice the .Rproj file. Double-click that file to re-open the project. Notice you get back to where you left off: it's the same working directory and command history, and all the files you were working on are still open. Because you followed my instructions above, you will, however, have a completely fresh environment, guaranteeing that you're starting with a clean slate.

9.1.4 File Organization

You should be saving all your files associated with your project in one directory. Here's a basic organization structure that I recommend:

```
~~~
masters_thesis:
  masters_thesis.Rproj
  01_Clean.R
  02_Model.R
  03_Visualizations.R
  Data/
    raw/
      un-raw.csv
      worldbank-raw.csv
    cleaned/
      country-year.csv
  Results:
    regressions
      h1.txt
      h2.txt
    figures
      bivariate.pdf
      bar_plot.pdf
~~~
```

Here are some important tips:

- read raw data from the `Data` subdirectory. Don't ever change or overwrite the raw data!
- export cleaned and altered data into a separate directory.
- write separate scripts for each stage in the research pipeline. Keep scripts short and focused on one main purpose. If a script gets too long, that might be a sign you need to split it up.
- write scripts that reproduce your results and figures, and write them in the `Results` subdirectory.

Acknowledgements

This page is in part derived from the following sources:

1. R for Data Science licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0

More Resources

- Gentzkow, Matthew and Jesse M. Shapiro. 2014. Code and Data for the Social Sciences: A Practitioner's Guide.

9.2 Importing and Exporting

9.2.1 Where's my data?

To start, you first need to know where your data lives. Sometimes, the data is stored as a file on your computer, e.g. csv, Excel, SPSS, or some other file type. When the data is on your computer, we say the data is stored **locally**.

Data can also be stored externally on the Internet, in a package, or obtained through other sources. For example, some R packages contain datasets (like the `gapminder` package). Later in this course, we'll discuss how to obtain data from web APIs and websites. For now, the rest of the unit discusses data that is stored **locally**.

9.2.2 Data Storage

Ideally, your data should be stored in a certain file format. I recommend a `csv` (comma separated value) file, which formats spreadsheet (rectangular) data in a plain-text format. `csv` files are plain-text, and can be read into almost any statistical software program, including R. Try to avoid Excel files if you can.

Here are some other tips:

- When working with spreadsheets, the first row is usually reserved for the header, while the first column is used to identify the sampling unit (**unique identifier**, or **key**);
- Avoid file names and variable names with blank spaces. This can cause errors when reading in data.
- If you want to concatenate words, insert a `.` or `_` in between two words instead of a space;
- Short names are prefered over longer names;

- Try to avoid using names that contain symbols such as `?`, `$`,`%`, `^`, `&`, `*`, `(`, `)`, `-`, `#`, `?`, `,`, `<`, `>`, `/`, `|`, `\`, `[` ,`]`,`{`, and `}`;
- make sure that any missing values in your data set are indicated with `NA` or blank fields (don't use 99 or 77).

9.2.3 Importing Data

Find Paths First

In order to import (or read) data into R, you first have to know where it is, and how to find it.

First, remember that you'll need to know the *current working directory* so that you know where R is looking for files. If you're using R Projects, that working directory will be the top-level directory of the project.

Second, you'll need to know where the data file is, relative to your working directory. If it's stored in the `Data/raw/` folder, the relative path to your file will be `Data/raw/file-name.csv`

Reading Tabular Data

The workhorse for reading into a data frame is *read.table()*, which allows any separator (CSV, tab-delimited, etc.). *read.csv()* is a special case of *read.table()* for CSV files.

The basic formula is:

```
# Basic CSV read: Import data with header row, values separated by "," decimals as "."
mydataset <- read.csv(file="  ", stringsAsFactors=)
```

Here's a practical example, using the PolityIV dataset:

```
# import polity
polity <- read.csv("data/polity_sub.csv", stringsAsFactors = F)
head(polity)
#>       country year polity2
#> 1 Afghanistan 1800      -6
#> 2 Afghanistan 1801      -6
#> 3 Afghanistan 1802      -6
#> 4 Afghanistan 1803      -6
#> 5 Afghanistan 1804      -6
#> 6 Afghanistan 1805      -6
```

We use `stringsAsFactors = F` in order to treat text columns as character vectors, not as factors. If we don't set this, the default is that all non-numerical columns will be encoded as factors. This behavior usually makes poor sense,

and is due to historical reasons. At one point in time, factors were faster than character vectors, so R's `read.table()` set the default to read in text as factors.

`read.table()` has a number of other options:

```
# For importing tabular data with maximum customizeability
mydataset <- read.table(file=, header=, sep=, quote=, dec=, fill=, stringsAsFactors=)
```

You might also see commands like `read_csv()` (notice the underscore insteadof the period). This is the tidyverse version of `read.csv()` accomplishes the same task.

Reading Excel Files

Don't use Microsoft Excel files (.xls or .xlsx). But if you must:

```
# Make sure you have installed the tidyverse suite (only necessary one time)
# install.packages("tidyverse") # Not Run

# Load the "readxl" package (necessary every new R session)
library(readxl)
```

`read_excel()` reads both `xls` and `xlsx` files and detects the format from the extension.

```
# Basic call
mydataset <- read_excel(path = , sheet = ")
```

Here's a real example:

```
# Example with .xlsx (single sheet)
air <- read_excel("data/airline_small.xlsx", sheet = 1)
air[1:5, 1:5]
#> # A tibble: 5 x 5
#>    Year Month DayofMonth DayOfWeek DepTime
#>   <dbl> <dbl>      <dbl>     <dbl> <chr>
#> 1  2005    11         22         2 1700
#> 2  2008     1         31         4 2216
#> 3  2005     7         17         7 905
#> 4  2008     9         23         2 859
#> 5  2005     3          5         6 827
```

Reading Stata (.dta) Files

There are many ways to read `.dta` files into R. I recommend using `haven` because it is part of the `tidyverse`.

```
library(haven)
air.dta <- read_dta("data/airline_small.dta")
air[1:5, 1:5]
#> # A tibble: 5 x 5
#>    Year Month DayofMonth DayOfWeek DepTime
#>   <dbl> <dbl>      <dbl>     <dbl> <chr>
#> 1  2005    11         22         2 1700
#> 2  2008     1         31         4 2216
#> 3  2005     7         17         7 905
#> 4  2008     9         23         2 859
#> 5  2005     3          5         6 827
```

For Really Big Data

If you have really big data, `read.csv()` will be too slow. In these cases, check out the following options:

1) `read_csv()` in the `readr` package is a faster, more helpful drop-in replacement for `read.csv()` that plays well with tidyverse packages (discussed in future lessons).
2) the `data.table` package is great for reading and manipulating large datasets (orders of gigabytes or 10s of gigabytes)

9.2.4 Exporting Data

You should never go from raw data to results in one script. Typically, you'll want to import raw data, clean it, and then export that cleaned dataset onto your computer. That cleaned dataset will then be imported into another script for analysis, in a modular fashion.

To export (or write) data from R onto your computer, you can create individual `.csv` files, or export many data objects into an `.RData` object.

Writing a `csv` Spreadsheet

To export an individual dataframe as a spreadsheet, use `write.csv()`

```
# Basic call
write.csv(x = , file = , row.names = , col.names =)
```

Let's write the `air` dataset as a csv.

```
# Basic call
write.csv(air, "data/airlines.csv", row.names = F)
```

Packaging Data into .RData

Sometimes, it's helpful to write several dataframes at once, to be used in later analysis. To do so, we use the `save()` function to create one file containing many R data objects.

```
# Basic call
save(..., file = )
```

Here's how we can write both `air` and `polity` into one file.

```
save(air, polity, file = "data/datasets.RData")
```

We can then read these datasets back into R using `load()`

```
# clear environment
rm(list=ls())

# load datasets
load("data/datasets.RData")
```

Acknowledgements

This page is in part derived from the following sources:

1. R for Data Science licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0
2. Gentzkow, Matthew and Jesse M. Shapiro. 2014. Code and Data for the Social Sciences: A Practitioner's Guide.

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

Chapter 10

Data Transformation

10.1 Introduction to Data

The upcoming weeks will be focused on using R for data cleaning and analysis. Let's first get on the same page with some terms:

- A **variable** is a quantity, quality, or property that you can measure.
- An **observation** is a set of measurements for the same unit. An observation will contain several values, each associated with a different variable. I'll sometimes refer to an observation as a **data point** or an **element**.
- A **value** is the state of a variable for a particular observation.
- **Tabular data** is a set of values, each associated with a variable and an observation. Tabular data has rows (observations) and columns (variables). Also called **rectangular** data or **spreadsheets**.

10.1.1 The Gapminder Dataset

This lesson discusses how to perform basic exploratory data analysis.

For this unit, we'll be working with the "Gapminder" dataset, which is an excerpt of the data available at Gapminder.org. For each of 142 countries, the data provides values for life expectancy, GDP per capita, and population, every five years, from 1952 to 2007.

```
require(gapminder)
gap <- gapminder
kable(head(gap))
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.8	8425333	779
Afghanistan	Asia	1957	30.3	9240934	821
Afghanistan	Asia	1962	32.0	10267083	853
Afghanistan	Asia	1967	34.0	11537966	836
Afghanistan	Asia	1972	36.1	13079460	740
Afghanistan	Asia	1977	38.4	14880372	786

10.1.2 Structure and Dimensions

The first things we want to know about a dataset are its dimensions and basic structure. For instance, we can look at the number of rows and columns:

```
# get number of rows and columns:
dim(gap)
#> [1] 1704    6
```

We might also want to see the names of the columns:

```
# see column names
names(gap)
#> [1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

The `str` function is helpful to see an overview of the data's structure:

```
# see structure of data
str(gap)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
#>  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
#>  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 138
#>  $ gdpPercap: num  779 821 853 836 740 ...
```

Finally, I encourage you to actually peak at the data itself. The `head` function displays the first 6 rows of any dataframe.

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

```
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
```

10.1.3 Summary statistics

We can get quick summary statistics using `summary`. Passing the entire dataframe will summarize all columns:

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

Passing a column with summarize that particular column:

```
summary(gap$year)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1952    1966    1980    1980    1993    2007
```

Sometimes we need to do some basic checking for the number of observations or types of observations in our dataset. To do this quickly and easily, `table()` is our friend.

Let's look at the number of observations first by region, and then by both region and year.

```
table(gap$continent)
#>
#>   Africa Americas     Asia   Europe  Oceania
#>      624      300      396      360       24
table(gap$continent, gap$year)
#>
#>             1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
#>   Africa      52   52   52   52   52   52   52   52   52   52   52   52
#>   Americas    25   25   25   25   25   25   25   25   25   25   25   25
```

```
#>   Asia       33   33   33   33   33   33   33   33   33   33   33   33
#>   Europe     30   30   30   30   30   30   30   30   30   30   30   30
#>   Oceania     2    2    2    2    2    2    2    2    2    2    2    2
```

We can even divide by the total number of rows to get proportion, percent, etc.

```
table(gap$continent)/nrow(gap)
#>
#>   Africa Americas     Asia   Europe  Oceania
#>   0.3662   0.1761   0.2324   0.2113   0.0141
table(gap$continent)/nrow(gap)*100
#>
#>   Africa Americas     Asia   Europe  Oceania
#>    36.62    17.61    23.24    21.13     1.41
```

10.1.4 Challenges

Challenge 1.

Read the `polity_sub` dataset in the `Data` sub-directory.

Challenge 2.

Report the number and names of each variable in the dataset.

Challenge 3.

What is the mean `polity2` score in the dataset?

Challenge 4.

What is the range of the polity2 variable?

Challenge 5.

How many unique countries are in the dataset?

10.2 Introduction to Tidyverse

10.2.1 `tidyverse`

> It is often said that 80% of data analysis is spent on the process of cleaning and preparing the data. (Dasu and Johnson, 2003)

For most applied researchers, data preparation usually involves 3 main steps.

1. ***Transforming*** data frames, e.g. filtering, summarizing, and conducting calculations across groups.
2. ***Tidying*** data into the appropriate format.
3. ***Merging*** or linking several datasets to create a bigger dataset.

The `tidyverse` is a suite of packages designed specifically to help with these steps. These are by no means the only packages out there for data wrangling, but they are increasingly popular for their readable, straightforward syntax and sensible default behaviors.

In this chapter, we're going to focus on how to use the `dplyr` package for data transformation tasks.

10.2.2 Gapminder

For this unit, we'll be working with the "Gapminder" dataset again.

```
gap <- gapminder
kable(head(gap))
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.8	8425333	779
Afghanistan	Asia	1957	30.3	9240934	821
Afghanistan	Asia	1962	32.0	10267083	853
Afghanistan	Asia	1967	34.0	11537966	836
Afghanistan	Asia	1972	36.1	13079460	740
Afghanistan	Asia	1977	38.4	14880372	786

10.2.3 Why `dplyr`?

So far, you've seen the basics of manipulating data frames, e.g. subsetting and basic calculations. For instance, we can use base R functions to calculate summary statistics across groups of observations, e.g., the mean GDP per capita within each region:

```
mean(gap$gdpPercap[gap$continent == "Africa"])
#> [1] 2194
mean(gap$gdpPercap[gap$continent == "Americas"])
```

```
#> [1] 7136
mean(gap$gdpPercap[gap$continent == "Asia"])
#> [1] 7902
```

But this isn't ideal because it involves a fair bit of repetition. Repeating yourself will cost you time, both now and later, and potentially introduce some nasty bugs.

Luckily, the `dplyr` package provides a number of very useful functions for manipulating dataframes. These functions will save you time by reducing repetition. As an added bonus, you might even find the `dplyr` grammar easier to read.

Here, we're going to cover 7 of the most commonly used `dplyr` functions. We'll also cover pipes (`%>%`), which are used to combine them.

1. `select()`
2. `filter()`
3. `group_by()`
4. `summarize()`
5. `mutate()`
6. `arrange()`
7. `count()`

If you have have not installed tidyverse, please do so now:

```
# not run
# install.packages('tidyverse')
require(tidyverse)
```

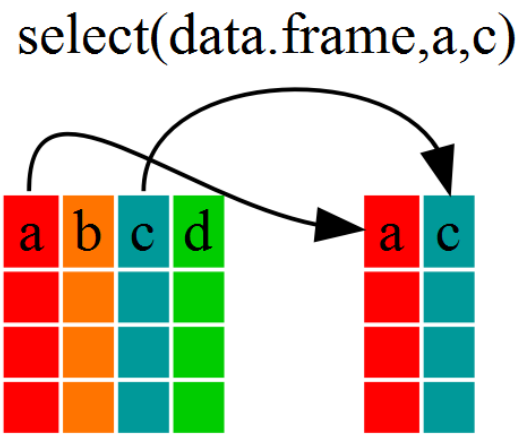
10.3 `dplyr` Functions

10.3.1 Select Columns with `select`

Imagine that we just received the gapminder dataset, but are only interested in a few variables in it. We could use the `select()` function to keep only the variables we select.

```
year_country_gdp <- select(gap, year, country, gdpPercap)
kable(head(year_country_gdp))
```

year	country	gdpPercap
1952	Afghanistan	779
1957	Afghanistan	821
1962	Afghanistan	853
1967	Afghanistan	836
1972	Afghanistan	740
1977	Afghanistan	786



If we open up `year_country_gdp`, we'll see that it only contains the year, country and gdpPercap. This is equivalent to the base R subsetting function:

```
year_country_gdp_base <- gap[,c("year", "country", "gdpPercap")]
kable(head(year_country_gdp))
```

year	country	gdpPercap
1952	Afghanistan	779
1957	Afghanistan	821
1962	Afghanistan	853
1967	Afghanistan	836
1972	Afghanistan	740
1977	Afghanistan	786

We can also use select to rename columns:

```
year_country_gdp_named <- select(gap, Year = year, Country = country, GDP_per_capita = gdpPercap)
```

But, as we will see, `dplyr` makes for much more readable, efficient code because of its *pipe* operator.

10.3.2 The Pipe

Above, we used what's called 'normal' grammar, but the strengths of `dplyr` lie in combining several functions using *pipes*. Since the pipes grammar is unlike anything we've seen in R before, let's repeat what we've done above using pipes.

In typical base R code, a simple operation might be written like:

```
# NOT run
cupcakes <- bake(pour(mix(ingredients)))
```

A computer has no trouble understanding this and your cupcakes will be made just fine, but a person has to read right to left to understand the order of operations - the opposite of how most western languages are read - making it harder to understand what is being done!

To be more readable without pipes, we might break up this code into intermediate objects:

```
## NOT run
batter <- mix(ingredients)
muffin_tin <- pour(batter)
cupcakes <- bake(muffin_tin)
```

But, this can clutter our environment with a lot of variables that aren't very useful to us. Plus, these variables are often are named very similar things (e.g. step, step1, step2…) which can lead to confusion and the creation of hard-to-track-down bugs.

Enter the Pipe...

The *pipe* makes it easier to read code by laying out operations from left to right – each line can be read like a line of a recipe for the perfect data frame!

Pipes take the input on the left side of the `%>%` symbol and pass it in as the first argument to the function on the right side.

With pipes, our cupcake example might be written like:

```
## NOT run
cupcakes <- ingredients %>%
  mix() %>%
  pour() %>%
  bake()
```

Tips for Piping

1. Remember that you don't assign anything within the pipes – that is, you should not use <- inside the piped operation. Only use this at the beginning of your code if you want to save the output.
2. Remember to add the pipe `%>%` at the end of each line involved in the piped operation. A good rule of thumb: since RStudio will automatically indent lines of code that are part of a piped operation, if the line isn't indented, it probably hasn't been added to the pipe. If you have an error in a piped operation, always check to make sure the pipe is connected as you expect.
3. In RStudio, the hotkey for the pipe is Ctrl + Shift + M.

`select` & Pipe (`%>%`)

Since the pipe grammar is unlike anything we've seen in R before, let's repeat what we did above with the gapminder dataset using pipes:

```
year_country_gdp <- gap %>% select(year, country, gdpPercap)
```

Let's walk through it step by step.

First, we summon the gapminder data frame and pass it on to the next step using the pipe symbol `%>%`.

The second step is the `select()` function. In this case, we don't specify which data object we use in the call to `select()` since we've piped it in from the previous line.

Fun Fact: There is a good chance you have encountered pipes before in the shell. In R, a pipe symbol is `%>%` while in the shell it is `|`. But the concept is the same!

10.3.3 Filter Rows with `filter`

Now let's say we're only interested in African countries. We can combine `select` and `filter` to select only the observations where `continent` is `Africa`.

```
year_country_gdp_africa <- gap %>%
    filter(continent == "Africa") %>%
    select(year, country, gdpPercap)
```

As with last time, first we pass the gapminder dataframe to the `filter()` function, then we pass the filtered version of the gapminder dataframe to the `select()` function.

To clarify, both the `select` and `filter` functions subset the data frame. The difference is that `select` extracts certain columns, while `filter` extracts certain rows.

Note: The order of operations is very important in this case. If we used `select` first, filter would not be able to find the variable `continent` since we would have removed it in the previous step.

10.3.4 Calculate Across Groups with `group_by`

A common task you'll encounter when working with data is running calculations on different groups within the data. For instance, what if we wanted to calculate the mean GDP per capita for each continent?

In base R, you would have to run the `mean()` function for each subset of data:

```
mean(gap$gdpPercap[gap$continent == "Africa"])
#> [1] 2194
mean(gap$gdpPercap[gap$continent == "Americas"])
#> [1] 7136
mean(gap$gdpPercap[gap$continent == "Asia"])
#> [1] 7902
mean(gap$gdpPercap[gap$continent == "Europe"])
#> [1] 14469
mean(gap$gdpPercap[gap$continent == "Oceania"])
#> [1] 18622
```

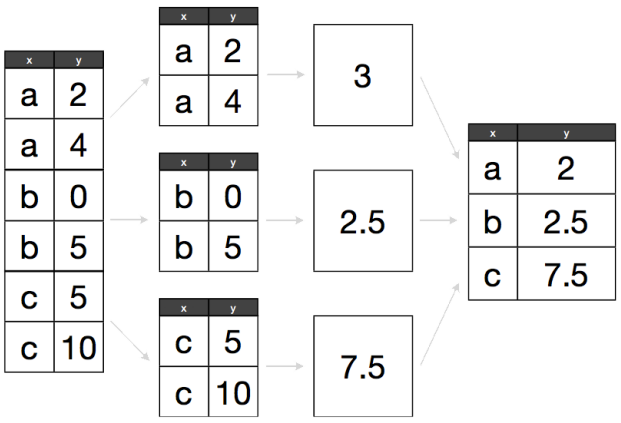
That's a lot of repetition! To make matters worse, what if we wanted to add these values to our original data frame as a new column? We would have to write something like this:

```
gap$mean.continent.GDP <- NA
gap$mean.continent.GDP[gap$continent == "Africa"] <- mean(gap$gdpPercap[gap$continent == "Africa'
gap$mean.continent.GDP[gap$continent == "Americas"] <- mean(gap$gdpPercap[gap$continent == "Ameri
gap$mean.continent.GDP[gap$continent == "Asia"] <- mean(gap$gdpPercap[gap$continent == "Asia"])
gap$mean.continent.GDP[gap$continent == "Europe"] <- mean(gap$gdpPercap[gap$continent == "Europe'
gap$mean.continent.GDP[gap$continent == "Oceania"] <- mean(gap$gdpPercap[gap$continent == "Oceani
```

You can see how this can get pretty tedious, especially if we want to calculate more complicated or refined statistics. We could use loops or apply functions, but these can be difficult, slow, and error-prone.

split-apply-combine

The abstract problem we're encountering here is know as "split-apply-combine":



We want to *split* our data into groups (in this case continents), *apply* some calculations on that group, then *combine* the results together afterwards.

Luckily, `dplyr` offers a much cleaner, straight-forward solution to this problem.

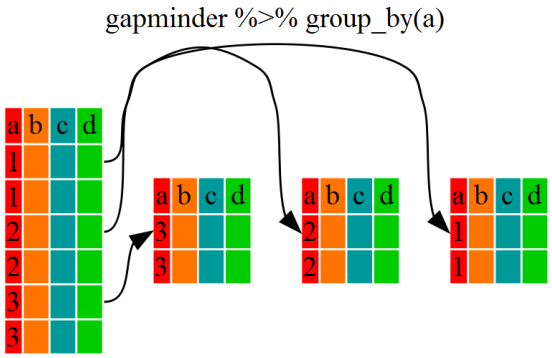
First, let's remove the column we just made.

```
gap <- gap %>% select(-mean.continent.GDP) # drop a column with -
# OR
gap$mean.continent.GDP <- NULL
```

10.3.4.1 `group_by`

We've already seen how `filter()` can help us select observations that meet certain criteria (in the above: `continent == "Africa"`). More helpful, however, is the `group_by()` function, which will essentially use every unique criteria that we could have used in `filter()`.

A `grouped_df` can be thought of as a `list` where each item in the `list` is a `data.frame` which contains only the rows that correspond to a particular value for `continent` (at least in the example above).



10.3.5 Summarize Across Groups with `summarize`

`group_by()` on its own is not particularly interesting. It's much more exciting used in conjunction with the `summarize()` function.

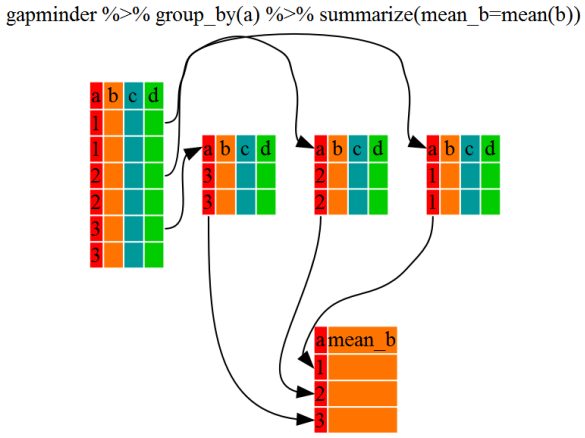
This will allow us to create new variable(s) by applying transformations to variables in each of our groups (continent-specific data frames).

In other words, using the `group_by()` function, we split our original data frame into multiple pieces, to which we then apply summary functions (e.g. `mean()` or `sd()`) within `summarize()`.

The output is a new data frame reduced in size, with one row per group.

```
gdp_bycontinents <- gap %>%
    group_by(continent) %>%
    summarize(mean_gdpPercap = mean(gdpPercap))
kable(head(gdp_bycontinents))
```

continent	mean_gdpPercap
Africa	2194
Americas	7136
Asia	7902
Europe	14469
Oceania	18622



That allowed us to calculate the mean gdpPercap for each continent.

But it gets even better – the function `group_by()` allows us to group by multiple variables. Let's group by `year` and `continent`.

```
gdp_bycontinents_byyear <- gap %>%
    group_by(continent, year) %>%
    summarize(mean_gdpPercap = mean(gdpPercap))
kable(head(gdp_bycontinents_byyear))
```

continent	year	mean_gdpPercap
Africa	1952	1253
Africa	1957	1385
Africa	1962	1598
Africa	1967	2050
Africa	1972	2340
Africa	1977	2586

That is already quite powerful, but it gets even better! You're not limited to defining 1 new variable in `summarize()`.

```
gdp_pop_bycontinents_byyear <- gap %>%
    group_by(continent, year) %>%
    summarize(mean_gdpPercap = mean(gdpPercap),
```

```
              sd_gdpPercap = sd(gdpPercap),
              mean_pop = mean(pop),
              sd_pop = sd(pop))
kable(head(gdp_pop_bycontinents_byyear))
```

continent	year	mean_gdpPercap	sd_gdpPercap	mean_pop	sd_pop
Africa	1952	1253	983	4570010	6317450
Africa	1957	1385	1135	5093033	7076042
Africa	1962	1598	1462	5702247	7957545
Africa	1967	2050	2848	6447875	8985505
Africa	1972	2340	3287	7305376	10130833
Africa	1977	2586	4142	8328097	11585184

10.3.6 Add New Variables with `mutate`

What if we wanted to add these values to our original data frame instead of creating a new object?

For this, we can use the `mutate()` function, which is similar to `summarize()` except that it creates new variables in the same dataframe that you pass into it.

```
gapminder_with_extra_vars <- gap %>%
    group_by(continent, year) %>%
    mutate(mean_gdpPercap = mean(gdpPercap),
              sd_gdpPercap = sd(gdpPercap),
              mean_pop = mean(pop),
              sd_pop = sd(pop))
kable(head(gapminder_with_extra_vars))
```

country	continent	year	lifeExp	pop	gdpPercap	mean_gdpPercap	sd_gdpPerc
Afghanistan	Asia	1952	28.8	8425333	779	5195	186
Afghanistan	Asia	1957	30.3	9240934	821	5788	195
Afghanistan	Asia	1962	32.0	10267083	853	5729	164
Afghanistan	Asia	1967	34.0	11537966	836	5971	140
Afghanistan	Asia	1972	36.1	13079460	740	8187	190
Afghanistan	Asia	1977	38.4	14880372	786	7791	118

We can use also use `mutate()` to create new variables prior to (or even after) summarizing the information.

```
gdp_pop_bycontinents_byyear <- gap %>%
    mutate(gdp_billion = gdpPercap*pop/10^9) %>%
    group_by(continent, year) %>%
    summarize(mean_gdpPercap = mean(gdpPercap),
              sd_gdpPercap = sd(gdpPercap),
              mean_pop = mean(pop),
```

```
              sd_pop = sd(pop),
              mean_gdp_billion = mean(gdp_billion),
              sd_gdp_billion = sd(gdp_billion))
kable(head(gdp_pop_bycontinents_byyear))
```

continent	year	mean_gdpPercap	sd_gdpPercap	mean_pop	sd_pop	mean_gdp_billion	sd_gdp_b
Africa	1952	1253	983	4570010	6317450	5.99	
Africa	1957	1385	1135	5093033	7076042	7.36	
Africa	1962	1598	1462	5702247	7957545	8.79	
Africa	1967	2050	2848	6447875	8985505	11.44	
Africa	1972	2340	3287	7305376	10130833	15.07	
Africa	1977	2586	4142	8328097	11585184	18.70	

`mutate` vs. `summarize`

It can be confusing to decide whether to use `mutate` or `summarize`. The key distinction is whether you want the output to have one row for each group or one row for each row in the original data frame:

- `mutate`: creates new columns with as many rows as the original data frame
- `summarize`: creates a data frame with as many rows as groups

Note that if you use an aggregation function such as `mean()` within `mutate()` without using `group_by()`, you'll simply do the summary over all the rows of the input data frame.

And if you use an aggregation function such as `mean()` within `summarize()` without using `group_by()`, you'll simply create an output data frame with one row (i.e., the whole input data frame is a single group).

10.3.7 Arrange Rows with `arrange`

Let's say we want to sort the rows in our data frame according to values in a certain column. We can use the `arrange()` function to do this. For instance, let's organize our rows by `year` (recent first), and then by `continent`.

```
gapminder_with_extra_vars <- gap %>%
    group_by(continent, year) %>%
    mutate(mean_gdpPercap = mean(gdpPercap),
              sd_gdpPercap = sd(gdpPercap),
              mean_pop = mean(pop),
              sd_pop = sd(pop)) %>%
    arrange(desc(year), continent)
kable(head(gapminder_with_extra_vars))
```

country	continent	year	lifeExp	pop	gdpPercap	mean_gdpPercap	sd_gdpPer
Algeria	Africa	2007	72.3	33333216	6223	3089	3
Angola	Africa	2007	42.7	12420476	4797	3089	3
Benin	Africa	2007	56.7	8078314	1441	3089	3
Botswana	Africa	2007	50.7	1639131	12570	3089	3
Burkina Faso	Africa	2007	52.3	14326203	1217	3089	3
Burundi	Africa	2007	49.6	8390505	430	3089	3

10.3.8 Count Variable Quantities with `count()`

Finally, let's say we want to examine if thenumber of countries covered in the gapminder data set varies between years. We can use `count()` to count the number of observations within a set of parameters we choose.

Below, we will specify that we want to `count()` the number of observations in each year of the data set.

```
gap_count <- gap %>%
  dplyr::count(year)

kable(head(gap_count))
```

year	n
1952	142
1957	142
1962	142
1967	142
1972	142
1977	142

We can confirm that each year in the data set contains the same number of observations. We can use similar syntax to answer other questions: for example, how many countries in each year have a GDP that is greater than $10,000 per capita?

```
gap_count_gdp <- gap %>%
  filter(gdpPercap >= 10000) %>%
  dplyr::count(year)

kable(head(gap_count_gdp))
```

year	n
1952	7
1957	12
1962	19
1967	22
1972	32
1977	41

10.4 Challenges

Challenge 1.

Use `dplyr` to create a data frame containing the median `lifeExp` for each continent.

Challenge 2.

Use `dplyr` to add a column to the gapminder dataset that contains the total population of the continent of each observation in a given year. For example, if the first observation is Afghanistan in 1952, the new column would contain the population of Asia in 1952.

Challenge 3.

Use `dplyr` to: (a) add a column called `gdpPercap_diff` that contains the difference between the observation's `gdpPercap` and the mean `gdpPercap` of the continent in that year, (b) arrange the dataframe by the column you just created, in descending order (so that the relatively richest country-years are listed first).

hint: You might have to `ungroup()` before you `arrange()`.

Acknowledgments

Some of these materials in this module were adapted from:

- Software Carpentry
- R bootcamp at UC Berkeley

Chapter 11

Tidying Data

Even before we conduct analyses or calculations, we need to put our data into the correct format. The goal here is to rearrange a messy dataset into one that is **tidy**.

The two most important properties of tidy data are:

1) Each column is a variable.
2) Each row is an observation.

Tidy data is easier to work with because you have a consistent way of referring to variables (as column names) and observations (as row indices). The data then becomes easier to manipulate, visualize, and model.

For more on the concept of *tidy* data, read Hadley Wickham's paper here.

11.1 Wide vs. Long Formats

> Tidy datasets are all alike, but every messy dataset is messy in its own way.
>
> **Hadley Wickham**

Tabular datasets can be arranged in many ways. For instance, consider the data below. Each dataset displays information on heart rates observed in individuals across three different time periods, but the data are organized differently in each table.

```
wide <- data.frame(
  name = c("Wilbur", "Petunia", "Gregory"),
  time1 = c(67, 80, 64),
  time2 = c(56, 90, 50),
```

```
  time3 = c(70, 67, 101)
)
kable(wide)
```

name	time1	time2	time3
Wilbur	67	56	70
Petunia	80	90	67
Gregory	64	50	101

```
long <- data.frame(
  name = c("Wilbur", "Petunia", "Gregory", "Wilbur", "Petunia", "Gregory", "Wilbur", "
  time = c(1, 1, 1, 2, 2, 2, 3, 3, 3),
  heartrate = c(67, 80, 64, 56, 90, 50, 70, 67, 10)
)
kable(long)
```

name	time	heartrate
Wilbur	1	67
Petunia	1	80
Gregory	1	64
Wilbur	2	56
Petunia	2	90
Gregory	2	50
Wilbur	3	70
Petunia	3	67
Gregory	3	10

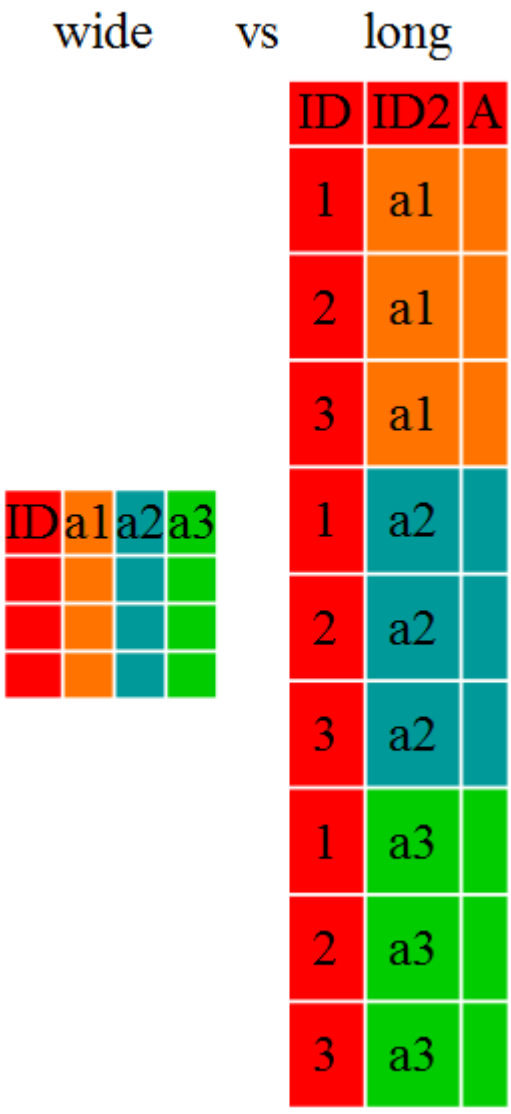
Question: Which one of these do you think is the *tidy* format?

Answer: The first dataframe (the "wide" one) would not be considered *tidy* because values (i.e., heart rate) are spread across multiple columns.

We often refer to these different structures as "long" vs. "wide" formats:

- In the **"long"** format, you usually have one column for the observed variable, and the other columns are ID variables.

- In the **"wide"** format, each row is often a site/subject/patient, and you have multiple observation variables containing the same type of data. These can be either repeated observations over time or observations of multiple variables (or a mix of both). In the case above, we had the same kind of data (heart rate) entered across three different columns, corresponding to three different time periods.



You may find data input in the "wide" format to be simpler, and some other applications may prefer "wide"-format data. However, many of R's functions have been designed assuming you have "long"-format data.

11.2 Tidying the Gapminder Data

Let's look at the structure of our original gapminder dataframe:

```
gap <- gapminder
kable(head(gap))
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.8	8425333	779
Afghanistan	Asia	1957	30.3	9240934	821
Afghanistan	Asia	1962	32.0	10267083	853
Afghanistan	Asia	1967	34.0	11537966	836
Afghanistan	Asia	1972	36.1	13079460	740
Afghanistan	Asia	1977	38.4	14880372	786

Question: Is this data frame **wide** or **long**?

Answer: This data frame is somewhere in between the purely 'long' and 'wide' formats. We have three "ID variables" (`continent`, `country`, `year`) and three "observation variables" (`pop`, `lifeExp`, `gdpPercap`).

Despite not having *all* observations in one column, this intermediate format makes sense given that all three observation variables have different units. As we have seen, many of the functions in R are often vector-based, and you usually do not want to do mathematical operations on values with different units.

On the other hand, there are some instances in which a purely long or wide format is ideal (e.g., plotting). Likewise, sometimes you will get data on your desk that is poorly organized, and you will need to **reshape** it.

11.3 `tidyr` Functions

Thankfully, the `tidyr` package will help you efficiently transform your data regardless of their original format.

```
# Load the "tidyverse" package (necessary every new R session):
require(tidyverse)
```

11.3.1 `gather`

Until now, we have been using the nicely formatted original gapminder dataset. This dataset is not quite wide and not quite long – it is something in the middle – but 'real' data (i.e., our own research data) will never be so well organized. Here let's start with the wide-format version of the gapminder dataset.

```
gap_wide <- read.csv("data/gapminder_wide.csv", stringsAsFactors = FALSE)
kable(head(gap_wide))
```

continent	country	gdpPercap_1952	gdpPercap_1957	gdpPercap_1962	gdpPercap_1967	gdpPerc
Africa	Algeria	2449	3014	2551	3247	
Africa	Angola	3521	3828	4269	5523	
Africa	Benin	1063	960	949	1036	
Africa	Botswana	851	918	984	1215	
Africa	Burkina Faso	543	617	723	795	
Africa	Burundi	339	380	355	413	

The first step towards getting our nice intermediate data format is to first convert from the wide to the long format.

The function `gather()` will 'gather' the observation variables into a single variable. This is sometimes called "melting" your data, because it melts the table from wide to long. Those data will be melted into two variables: one for the variable names and the other for the variable values.

```
gap_long <- gap_wide %>%
    gather(obstype_year, obs_values, 3:38)
kable(head(gap_long))
```

continent	country	obstype_year	obs_values
Africa	Algeria	gdpPercap_1952	2449
Africa	Angola	gdpPercap_1952	3521
Africa	Benin	gdpPercap_1952	1063
Africa	Botswana	gdpPercap_1952	851
Africa	Burkina Faso	gdpPercap_1952	543
Africa	Burundi	gdpPercap_1952	339

Notice that we put three arguments into the `gather()` function:

1. The name for the new ID variable (`obstype_year`).
2. The name for the new amalgamated observation variable (`obs_value`).
3. The indices of the old observation variables (`3:38`, signalling columns 3 through 38) that we want to gather into one variable. Notice that we do not want to melt down columns 1 and 2, as these are considered ID variables.

We can select observation variables using:

- Variable indices.
- Variable names (without quotes).
- `x:z` to select all variables between x and z.
- `-y` to *exclude* y.
- `starts_with(x, ignore.case = TRUE)`: All names that start with `x`.
- `ends_with(x, ignore.case = TRUE)`: All names that end with `x`.
- `contains(x, ignore.case = TRUE)`: All names that contain `x`.

See the `select()` function in `dplyr` for more options.

For instance, here we do the same thing with (1) the `starts_with` function and (2) the `-` operator:

```
# 1. With the starts_with() function:
gap_long <- gap_wide %>%
    gather(obstype_year, obs_values, starts_with('pop'),
           starts_with('lifeExp'), starts_with('gdpPercap'))
kable(head(gap_long))
```

continent	country	obstype_year	obs_values
Africa	Algeria	pop_1952	9279525
Africa	Angola	pop_1952	4232095
Africa	Benin	pop_1952	1738315
Africa	Botswana	pop_1952	442308
Africa	Burkina Faso	pop_1952	4469979
Africa	Burundi	pop_1952	2445618

```
# 2. With the - operator:
gap_long <- gap_wide %>%
  gather(obstype_year, obs_values, -continent, -country)
kable(head(gap_long))
```

continent	country	obstype_year	obs_values
Africa	Algeria	gdpPercap_1952	2449
Africa	Angola	gdpPercap_1952	3521
Africa	Benin	gdpPercap_1952	1063
Africa	Botswana	gdpPercap_1952	851
Africa	Burkina Faso	gdpPercap_1952	543
Africa	Burundi	gdpPercap_1952	339

However you choose to do it, notice that the output collapses all of the measured variables into two columns: one containing the new ID variable, the other containing the observation value for that row.

11.3.2 `separate`

You will notice that, in our long dataset, `obstype_year` actually contains two pieces of information, the observation type (`pop`, `lifeExp`, or `gdpPercap`) and the `year`.

We can use the `separate()` function to split the character strings into multiple variables.

```
gap_long_sep <- gap_long %>%
  separate(obstype_year, into = c('obs_type','year'), sep = "_") %>%
```

```
  mutate(year = as.integer(year))
kable(head(gap_long_sep))
```

continent	country	obs_type	year	obs_values
Africa	Algeria	gdpPercap	1952	2449
Africa	Angola	gdpPercap	1952	3521
Africa	Benin	gdpPercap	1952	1063
Africa	Botswana	gdpPercap	1952	851
Africa	Burkina Faso	gdpPercap	1952	543
Africa	Burundi	gdpPercap	1952	339

11.3.3 spread

The opposite of `gather()` is `spread()`. It spreads our observation variables back out to make a wider table. We can use this function to spread our `gap_long()` to the original "medium" format.

```
gap_medium <- gap_long_sep %>%
  spread(obs_type, obs_values)
kable(head(gap_medium))
```

continent	country	year	gdpPercap	lifeExp	pop
Africa	Algeria	1952	2449	43.1	9279525
Africa	Algeria	1957	3014	45.7	10270856
Africa	Algeria	1962	2551	48.3	11000948
Africa	Algeria	1967	3247	51.4	12760499
Africa	Algeria	1972	4183	54.5	14760787
Africa	Algeria	1977	4910	58.0	17152804

All we need is some quick fixes to make this dataset identical to the original `gapminder` dataset:

```
gap <- gapminder
kable(head(gap))
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.8	8425333	779
Afghanistan	Asia	1957	30.3	9240934	821
Afghanistan	Asia	1962	32.0	10267083	853
Afghanistan	Asia	1967	34.0	11537966	836
Afghanistan	Asia	1972	36.1	13079460	740
Afghanistan	Asia	1977	38.4	14880372	786

```
# Rearrange columns:
gap_medium <- gap_medium %>%
  select(country, continent, year, lifeExp, pop, gdpPercap)
kable(head(gap_medium))
```

country	continent	year	lifeExp	pop	gdpPercap
Algeria	Africa	1952	43.1	9279525	2449
Algeria	Africa	1957	45.7	10270856	3014
Algeria	Africa	1962	48.3	11000948	2551
Algeria	Africa	1967	51.4	12760499	3247
Algeria	Africa	1972	54.5	14760787	4183
Algeria	Africa	1977	58.0	17152804	4910

```
# Arrange by country, continent, and year:
gap_medium <- gap_medium %>%
  arrange(country,continent,year)
kable(head(gap_medium))
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.8	8425333	779
Afghanistan	Asia	1957	30.3	9240934	821
Afghanistan	Asia	1962	32.0	10267083	853
Afghanistan	Asia	1967	34.0	11537966	836
Afghanistan	Asia	1972	36.1	13079460	740
Afghanistan	Asia	1977	38.4	14880372	786

What we just told you will become obsolete...

`gather` and `spread` are being replaced by `pivot_longer` and `pivot_wider` in `tidyr 1.0.0`, which uses ideas from the `cdata` package to make reshaping easier to think about. In future classes, we will migrate to those functions.

11.4 More `tidyverse`

`dplyr` and `tidyr` have many more functions to help you wrangle and manipulate your data. See the Data Wrangling Cheatsheet for more.

There are some other useful packages in the tidyverse:

- `ggplot2` for plotting (we will cover this in the Visualization module).
- `readr` and `haven` for reading in data.
- `purrr` for working iterations.
- `stringr`, `lubridate`, and `forcats` for manipulating strings, dates, and factors, respectively.
- Many many more! Take a peak at the tidyverse github page...

11.5 Challenges

Challenge 1.

Subset the results from Challenge #3 (of the previous chapter) to select only the `country`, `year`, and `gdpPercap_diff` columns. Use `tidyr` to put it in wide format so that countries are rows and years are columns.

Challenge 2.

Now turn the dataframe above back into the long format with three columns: `country`, `year`, and `gdpPercap_diff`.

Acknowledgments

Some of the materials in this module were adapted from:

- Software Carpentry.
- R bootcamp at UC Berkeley.

Chapter 12

Relational Data

It is rare that data analysis involves only a single table of data. Typically, you have many tables of data, and you must combine them to answer the questions that you are interested in. Collectively, multiple tables of data are called **relational data** because it is the relations, not just the individual datasets, that are important.

Note that when we say relational database here, we are referring to how the data are structured, not to the use of any fancy software.

12.1 Why Relational Data

As social scientists, we are often working with data across different levels of analysis. The main principle of relational data is that each table is structured around the same observational unit.

Why is this important? Check out the following data.

```
messy <- data.frame(
  county = c(36037, 36038, 36039, 36040, NA , 37001, 37002, 37003),
  state = c('NY', 'NY', 'NY', NA, NA, 'VA', 'VA', 'VA'),
  cnty_pop = c(3817735, 422999, 324920, 143432, NA, 3228290, 449499, 383888),
  state_pop = c(43320903, 43320903, NA, 43320903, 43320903, 7173000, 7173000, 7173000),
  region = c(1, 1, 1, 1, 1, 3, 3, 4)
)

kable(messy)
```

county	state	cnty_pop	state_pop	region
36037	NY	3817735	43320903	1
36038	NY	422999	43320903	1
36039	NY	324920	NA	1
36040	NA	143432	43320903	1
NA	NA	NA	43320903	1
37001	VA	3228290	7173000	3
37002	VA	449499	7173000	3
37003	VA	383888	7173000	4

What a mess! How can the population of the state of New York be 43 million for one county but "missing" for another? If this is a dataset of counties, what does it mean when the "county" field is missing? If region is something like Census region, how can two counties in the same state be in different regions? And why is it that all the counties whose codes start with 36 are in New York except for one, where the state is unknown?

If we follow the principles of relational data, each type of observational unit should form a table:

- `counties` contains data on counties.
- `states` contains data on states.

So our data should look like this:

```
counties <- data.frame(
  county = c(36037, 36038, 36039, 36040, 37001, 37002, 37003),
  state = c('NY', 'NY', 'NY', 'NY', 'VA', 'VA', 'VA'),
  county_pop = c(3817735, 422999, 324920, 143432, 3228290, 449499, 383888), stringsAsFa
)
kable(counties)
```

county	state	county_pop
36037	NY	3817735
36038	NY	422999
36039	NY	324920
36040	NY	143432
37001	VA	3228290
37002	VA	449499
37003	VA	383888

```
states <- data.frame(
  state = c("NY", "VA"),
  state_pop = c(43320903, 7173000),
  region = c(1, 3), stringsAsFactors = F
)
kable(states)
```

state	state_pop	region
NY	43320903	1
VA	7173000	3

County population is a property of a county, so it lives in the county table. State population is a property of a state, so it cannot live in the county table. If we had panel data on counties, we would need separate tables for things that vary at the county level (like state) and things that vary at the county-year level (like population).

Now the ambiguity is gone. Every county has a population and a state. Every state has a population and a region. There are no missing states, no missing counties, and no conflicting definitions. The database is self-documenting.

12.2 Keys

The variables used to connect each pair of tables are called **keys**. A key is a variable (or set of variables) that uniquely identifies an observation; it can also be called a *unique identifier*.

- Keys are complete. They never take on missing values.
- Keys are unique. They are never duplicated across rows of a table.

In simple cases, a single variable is sufficient to identify an observation. In the example above, each county is identified with **county** (a numeric identifier); each state is identified with **state** (a two-letter string).

There are two types of keys:

- A **primary key** uniquely identifies an observation in its own table. For example, `counties$county` is a primary key because it uniquely identifies each county in the counties table.

- A **foreign key** uniquely identifies an observation in another table. For example, `counties$state` is a foreign key because it appears in the counties table where it matches each county to a unique state.

A primary key and the corresponding foreign key in another table form a **relation**.

Sometimes a table does not have an explicit primary key: each row is an observation, but no combination of variables reliably identifies it. If a table lacks a primary key, it is useful to add one with `mutate()` and `row_number()`. This is called a **surrogate key**.

12.3 Joins

Data stored in the form we have outlined above is considered *normalized*. In general, we should try to keep data normalized as far into the code pipeline as we can. Storing normalized data means your data will be easier to understand and it will be harder to make costly mistakes.

At some point, however, we are going to have to merge (or **join**) the tables together to produce and analyze a single dataframe.

Let's say we wanted to merge tables `x` and `y`. **join** allows us to combine variables from the two tables. It first matches observations by their keys, then copies across variables from one table to the other.

There are five join options:

1. An **inner join** keeps observations that appear in both tables.
2. A **left join** keeps all observations in `x`.
3. A **right join** keeps all observations in `y`.
4. A **full join** keeps all observations in `x` and all observations in `y`.
5. An **anti join** keeps all observations in `x` that do not have a match in `y`.

The most commonly used join is the `left_join()`: you use this whenever you look up additional data from another table, because it preserves the original observations even when there is not a match. For example, a `left_join()` on `x` and `y` pulls in variables from `y` while preserving all the observations in `x`.

Let's say we want to combine the `countries` and `states` tables we created earlier.

```
counties_states <- counties %>%
  left_join(states, by = "state")

kable(counties_states)
```

county	state	county_pop	state_pop	region
36037	NY	3817735	43320903	1
36038	NY	422999	43320903	1
36039	NY	324920	43320903	1
36040	NY	143432	43320903	1
37001	VA	3228290	7173000	3
37002	VA	449499	7173000	3
37003	VA	383888	7173000	3

Notice there are two new columns: `state_pop` and `region`.

The left join should be your default join: use it unless you have a strong reason to prefer one of the others.

12.4 Defining Keys

In the example above, the two tables were joined by a single variable, and that variable has the same name in both tables. That constraint was encoded by `by = "key"`.

You can use other values for `by` to connect the tables in other ways:

1. The default, **`by = NULL`**, uses all variables that appear in both tables, what we might call a "natural join."

For example, let's say we wanted to add a column to the `gapminder` dataset that encodes the regime type of each country-year observation. We will get that data from the `politiyIV` dataset.

```
gap <- gapminder
polity <- read.csv("data/polity_sub.csv", stringsAsFactors = F)
kable(head(polity))
```

country	year	polity2
Afghanistan	1800	-6
Afghanistan	1801	-6
Afghanistan	1802	-6
Afghanistan	1803	-6
Afghanistan	1804	-6
Afghanistan	1805	-6

We are now ready to join the tables. The common keys between them are `country` and `year`:

```
gap1 <- gapminder %>%
  left_join(polity)
#> Joining, by = c("country", "year")
#> Warning: Column `country` joining factor and character vector, coercing into
#> character vector

kable(head(gap1))
```

country	continent	year	lifeExp	pop	gdpPercap	polity2
Afghanistan	Asia	1952	28.8	8425333	779	-10
Afghanistan	Asia	1957	30.3	9240934	821	-10
Afghanistan	Asia	1962	32.0	10267083	853	-10
Afghanistan	Asia	1967	34.0	11537966	836	-7
Afghanistan	Asia	1972	36.1	13079460	740	-7
Afghanistan	Asia	1977	38.4	14880372	786	-7

2. A character vector, **`by = c("x", "y")`**. This is like a natural join, but it uses only some of the common variables.

3. A named character vector: `by = c("a" = "b")`. This will match variable `a` in table `x` to variable `b` in table `y`. The variables from `x` will be used in the output.

For example, let's add another variable to our `gapminder` dataset – physical integrity rights – from the CIRI dataset.

```
ciri <- read.csv("data/ciri_sub.csv", stringsAsFactors = F)
kable(head(ciri))
```

CTRY	YEAR	PHYSINT
Afghanistan	1981	0
Afghanistan	1982	0
Afghanistan	1983	0
Afghanistan	1984	0
Afghanistan	1985	0
Afghanistan	1986	0

Both datasets have country and year columns, but they are named differently.

```
gap2 <- gap1 %>%
  left_join(ciri, by = c("country" = "CTRY", "year" = "YEAR"))

kable(head(gap2))
```

country	continent	year	lifeExp	pop	gdpPercap	polity2	PHYSINT
Afghanistan	Asia	1952	28.8	8425333	779	-10	NA
Afghanistan	Asia	1957	30.3	9240934	821	-10	NA
Afghanistan	Asia	1962	32.0	10267083	853	-10	NA
Afghanistan	Asia	1967	34.0	11537966	836	-7	NA
Afghanistan	Asia	1972	36.1	13079460	740	-7	NA
Afghanistan	Asia	1977	38.4	14880372	786	-7	NA

Notice that `PHYSINT` is `NA` in the first 6 rows because the `ciri` dataset does not contain observations for Afghanistan in these years. But since we used `left_join()`, all observations in `gapminder` were preserved.

We can see some values for `PHYSINT` if we peak at the bottom of the dataset:

```
kable(tail(gap2))
```

country	continent	year	lifeExp	pop	gdpPercap	polity2	PHYSINT
Zimbabwe	Africa	1982	60.4	7636524	789	4	5
Zimbabwe	Africa	1987	62.4	9216418	706	-6	5
Zimbabwe	Africa	1992	60.4	10704340	693	-6	5
Zimbabwe	Africa	1997	46.8	11404948	792	-6	6
Zimbabwe	Africa	2002	40.0	11926563	672	-4	2
Zimbabwe	Africa	2007	43.5	12311143	470	-4	1

12.5 Duplicate Keys

So far we have assumed that the keys are unique, but that is not always the case. For example,

```
x <- data.frame(key = c(1, 2),
                val_y = c("x1", "x2"))

y <- data.frame(key = c(1, 2, 2, 1),
                val_x = c("y1", "y2", "y3", "y4"))

left_join(x, y, by = "key")
#>   key val_y val_x
#> 1   1    x1    y1
#> 2   1    x1    y4
#> 3   2    x2    y2
#> 4   2    x2    y3
```

Notice that this can sometimes cause unintended duplicates.

12.6 Challenges

Challenge 1.

Merge the polityVI and CIRI datasets, keeping all observations in polityVI. Save this merged dataframe as `p1`. How many observations does `p1` have? Why?

Challenge 2.

Merge the `gap1` dataset we created above with the `ciri` dataset, this time keeping all observations in `ciri`. Save this as `gap2`. How many observations does it have? What is the major problem with merging the datasets this way?

Acknowledgements

This page is in part derived from the following sources:

1. R for Data Science, licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0.
2. Gentzkow, Matthew and Jesse M. Shapiro. 2014. Code and Data for the Social Sciences: A Practitioner's Guide..

Chapter 13

Plotting

> "Make it informative, then make it pretty"

There are two major sets of tools for creating plots in R:

- 1. base, which come with all R installations.
- 2. ggplot2, a stand-alone package.

Note that other plotting facilities do exist (notably **lattice**), but base and `ggplot2` are by far the most popular.

13.1 The Dataset

For the following examples, we will be using the gapminder dataset we've used previously. Gapminder is a country-year dataset with information on life expectancy, among other things.

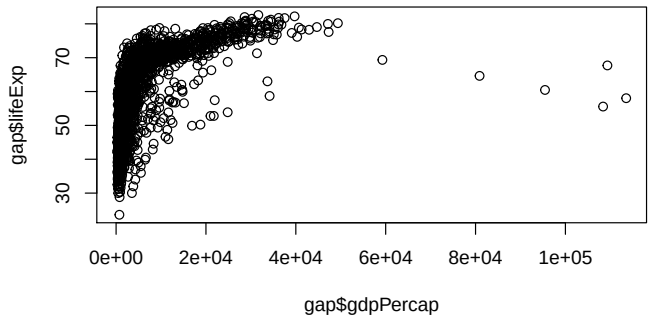
```
gap <- read.csv("data/gapminder-FiveYearData.csv", stringsAsFactors = F)
```

13.2 R Base Graphics

The **basic** call takes the following form:

```
plot(x=, y=)
```

```
plot(x = gap$gdpPercap, y = gap$lifeExp)
```



13.2.1 Scatter and Line Plots

The "type" argument accepts the following character indicators.

- "p" – point/scatter plots (default plotting behavior)

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p")
```

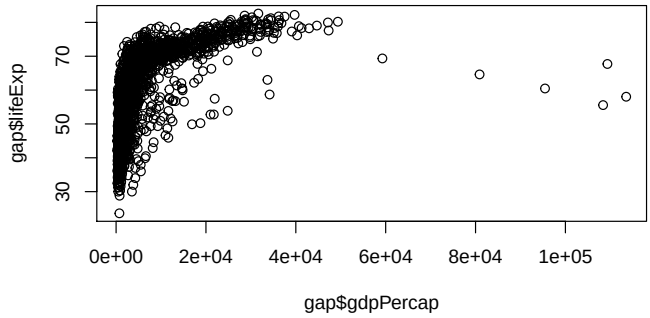


Figure 13.1:

- "l" – line graphs

```
# Note that "line" does not create a smoothing line, just connected points
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l")
```

- "b" – both line and point plots

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="b")
```

13.2.2 Histograms and Density Plots

Histograms display the frequency of different values of a variable.

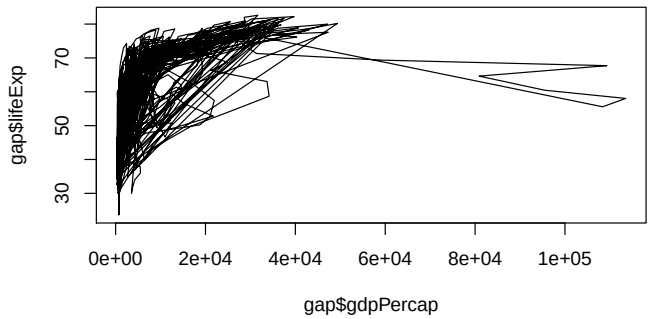


Figure 13.2:

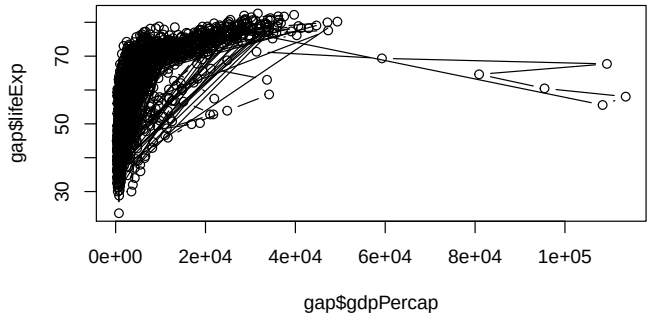
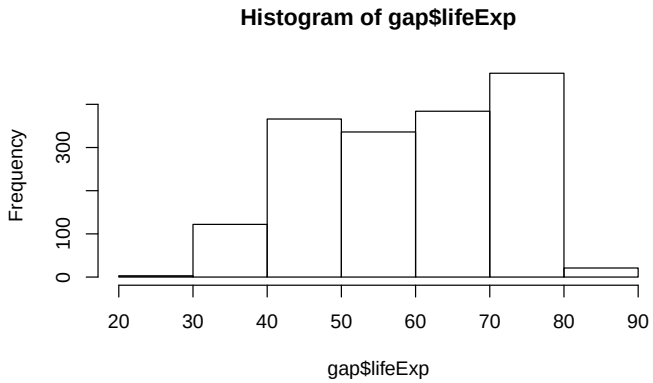


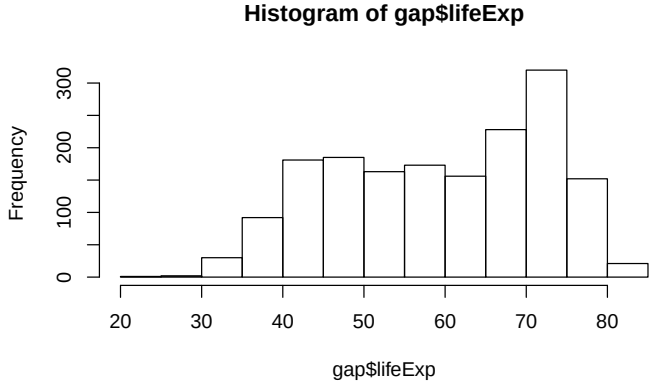
Figure 13.3:

```
hist(x=gap$lifeExp)
```

Histograms require a `breaks` argument, which determine the number of bins in the plot. Let's play around with different `breaks` values.

```
hist(x=gap$lifeExp, breaks=5)
hist(x=gap$lifeExp, breaks=10)
```





Density plots are similar, they visualize the distribution of data over a continu-

ous interval.

```
# Create a density object (NOTE: be sure to remove missing values)
age.density <- density(x=gap$lifeExp, na.rm=T)

# Plot the density object
plot(x=age.density)
```

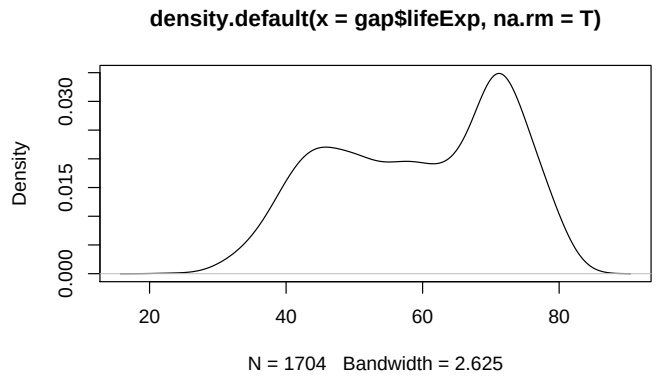
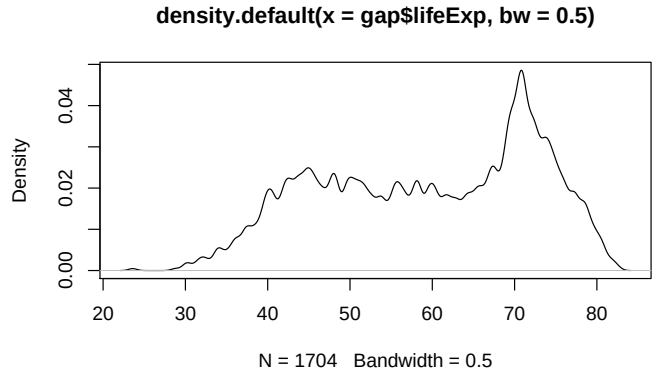
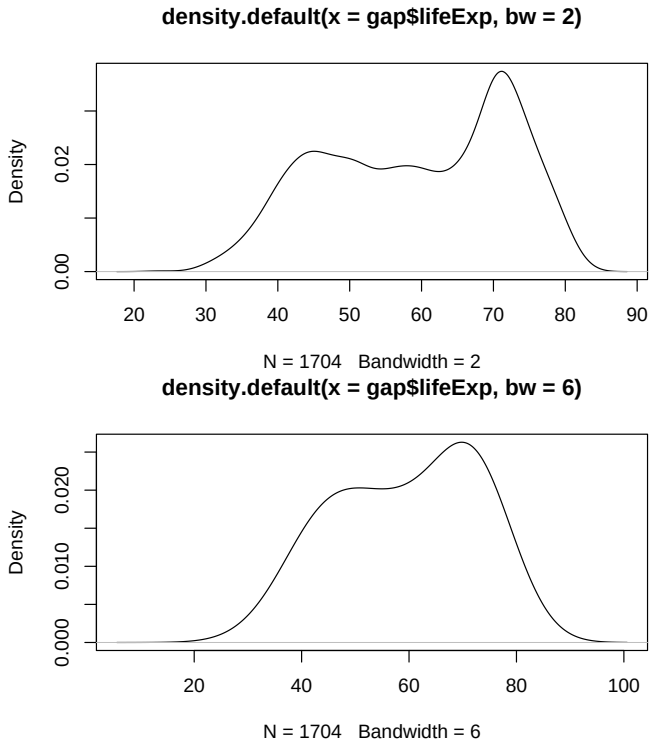


Figure 13.4:

Density passes a `bw` parameter, which determins the plot's "bandwidth".

```
# Plot the density object, bandwidth of 0.5
plot(x=density(x=gap$lifeExp, bw=.5))
# Plot the density object, bandwidth of 2
plot(x=density(x=gap$lifeExp, bw=2))
# Plot the density object, bandwidth of 6
plot(x=density(x=gap$lifeExp, bw=6))
```





13.2.3 Labels

Here's the basic call with popular labeling arguments:

```
plot(x=, y=, type="", xlab="", ylab="", main="")
```

From the previous example...

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", xlab="GDP per cap", ylab="Life Expe
```

13.2.4 Axis and Size Scaling

Currently it's hard to see the relationship between the points due to some strong outliers in GDP per capita. We can change the scale of units on the x-axis using scaling arguments.

Here's the basic call with popular scaling arguments

```
plot(x=, y=, type="", xlim=, ylim=, cex=)
```

From the previous example...

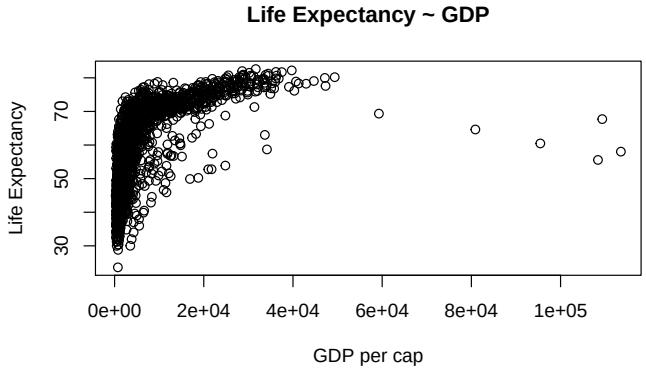


Figure 13.5:

```
# Create a basic plot
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p")
# Limit gdp (x-axis) to between 1,000 and 20,000
plot(x = gap$gdpPercap, y = gap$lifeExp, xlim = c(1000,20000))
# Limit gdp (x-axis) to between 1,000 and 20,000, increase point size to 2
plot(x = gap$gdpPercap, y = gap$lifeExp, xlim = c(1000,20000), cex=2)
# Limit gdp (x-axis) to between 1,000 and 20,000, decrease point size to 0.5
plot(x = gap$gdpPercap, y = gap$lifeExp, xlim = c(1000,20000), cex=0.5)
```

13.2.5 Graphical Parameters

We can change the points with a number of graphical options:

```
plot(x=, y=, type="", col="", pch=, lty=, lwd=)
```

- Colors

```
colors()[1:20] # View first 20 elements of the color vector
#>  [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
#>  [5] "antiquewhite2" "antiquewhite3" "antiquewhite4" "aquamarine"
#>  [9] "aquamarine1"   "aquamarine2"   "aquamarine3"   "aquamarine4"
#> [13] "azure"         "azure1"        "azure2"        "azure3"
#> [17] "azure4"        "beige"         "bisque"        "bisque1"
colors()[179] # View specific element of the color vector
#> [1] "gray26"
```

Another option: R Color Infographic

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", col=colors()[145]) # or col="gold3"
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", col="seagreen4") # or col=colors()[578]
```

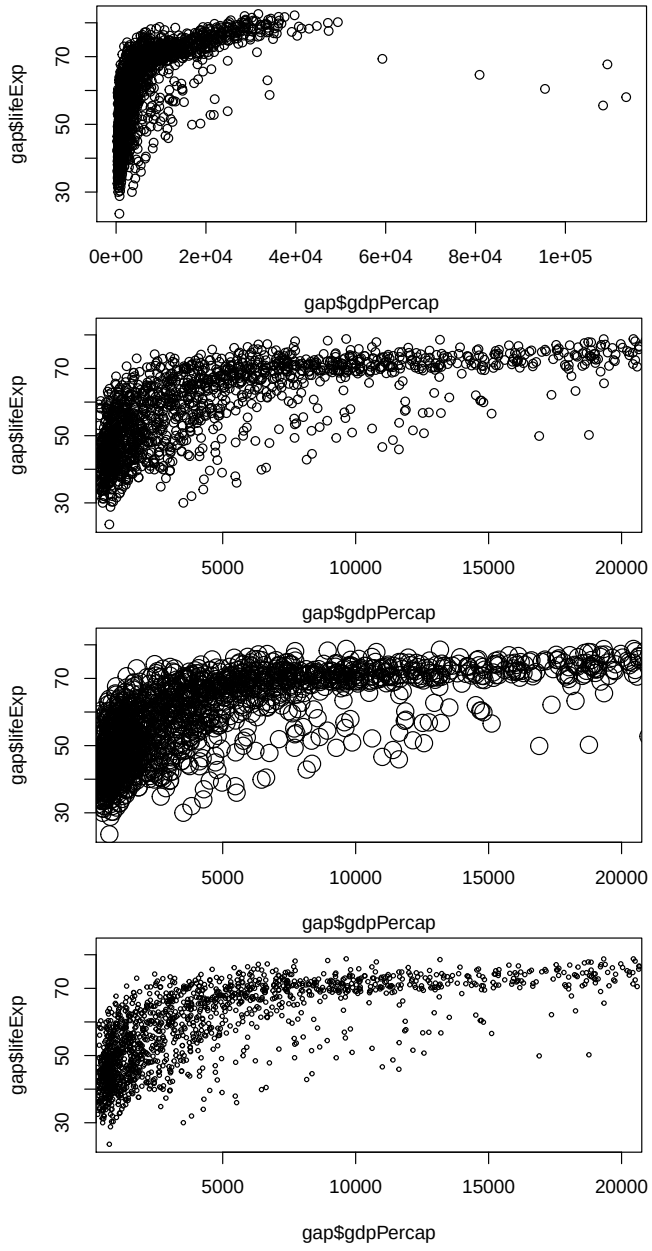

gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
5000
10000
15000
20000

Figure 13.6:

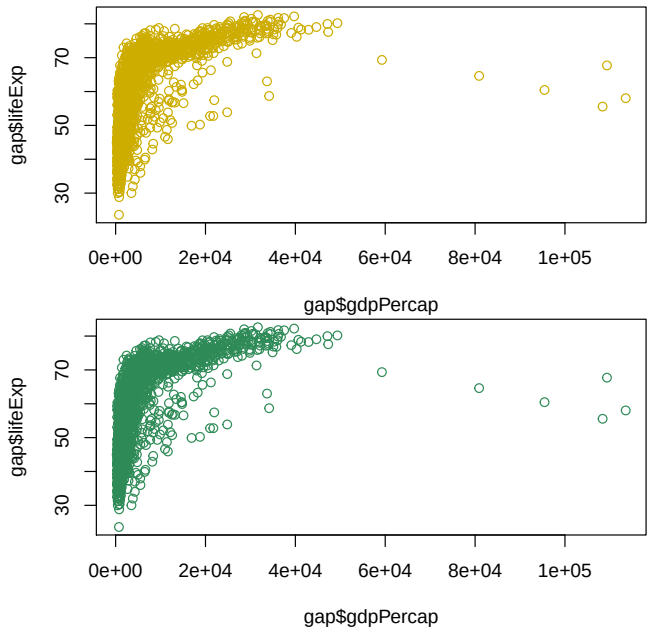


Figure 13.7:

- Point Styles and Widths

A Good Reference

```
# Change point style to crosses
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", pch=3)
# Change point style to filled squares
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p",pch=15)
# Change point style to filled squares and increase point size to 3
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p",pch=15, cex=3)
# Change point style to "w"
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", pch="w")
# Change point style to "$" and increase point size to 2
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p",pch="$", cex=2)
```

- Line Styles and Widths

```
# Line plot with solid line
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l", lty=1)
# Line plot with medium dashed line
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l", lty=2)
# Line plot with short dashed line
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l", lty=3)
# Change line width to 2
```

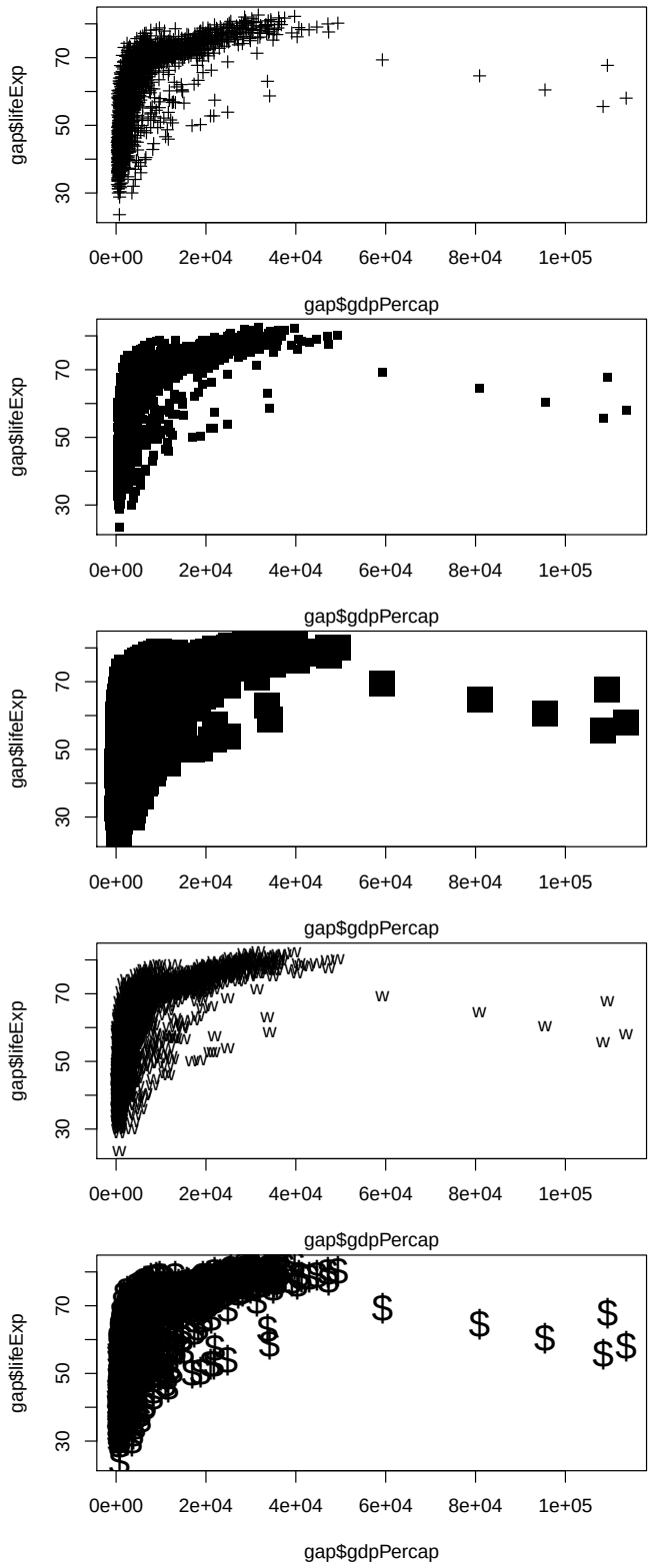

gap$lifeExp
gap$gdpPercap
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05

Figure 13.8:

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l", lty=3, lwd=2)
# Change line width to 5
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l",  lwd=5)
# Change line width to 10 and use dash-dot
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l",  lty=4, lwd=10)
```

13.2.6 Annotations, Reference Lines, and Legends

- Text

We can add text to an arbitrary point on the graph like this:

```
# plot the line first
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p")
# now add the label
text(x=40000, y=50, labels="Evens Out", cex = .75)
```

We can also add labels for every point by passing in a vector of text:

```
# first randomly select rows for a smaller gapaset
library(dplyr)
small <- gap %>% sample_n(100)

# plot the line first
plot(x = small$gdpPercap, y = small$lifeExp, type="p")
# now add the label
text(x = small$gdpPercap, y = small$lifeExp, labels = small$country)
```

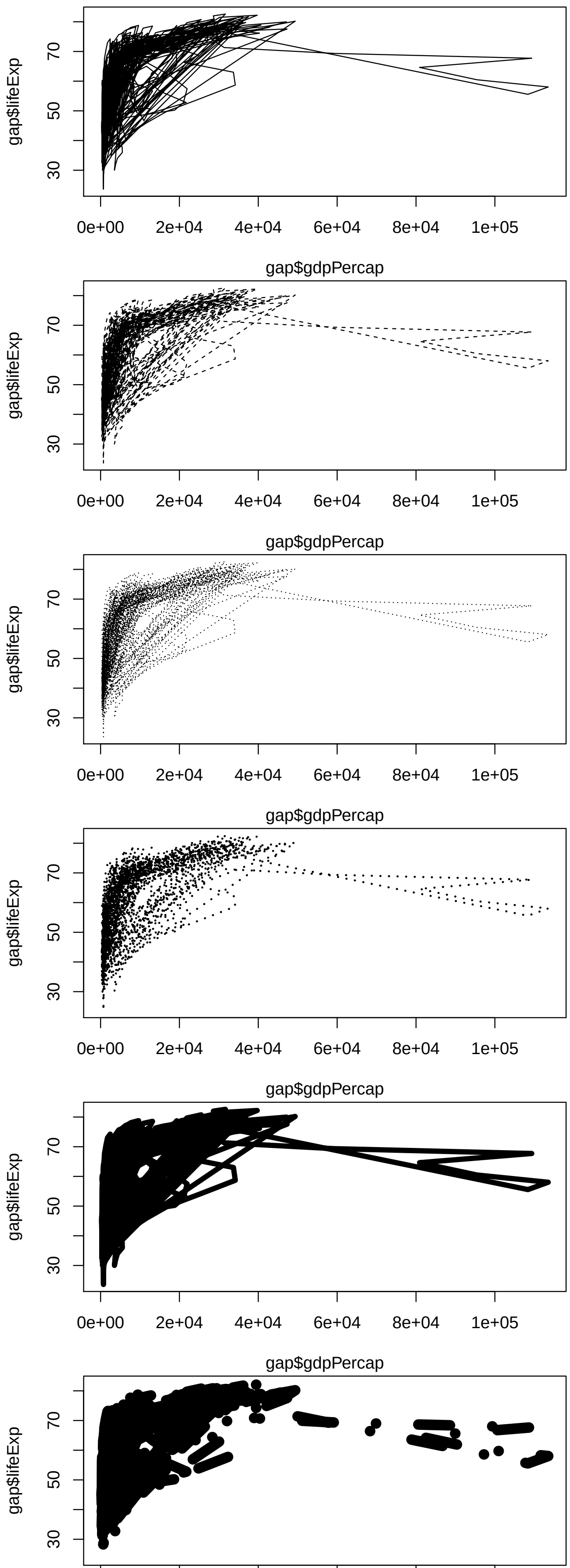
- Reference Lines

```
# plot the line
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p")
# now the guides
abline(v=40000, h=75, lty=2)
```

13.3 ggplot2

Setup:

```
library(ggplot2)
gap <- read.csv("data/gapminder-FiveYearData.csv", stringsAsFactors = F)
```


gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70

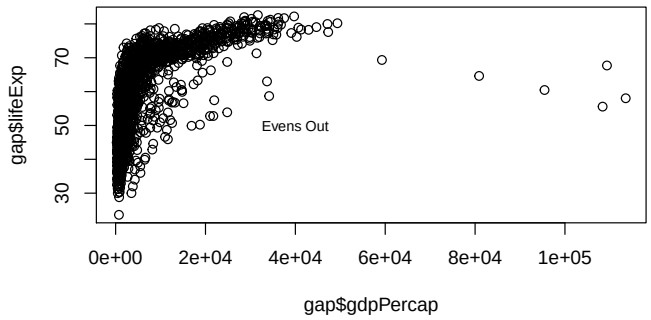


Figure 13.10:

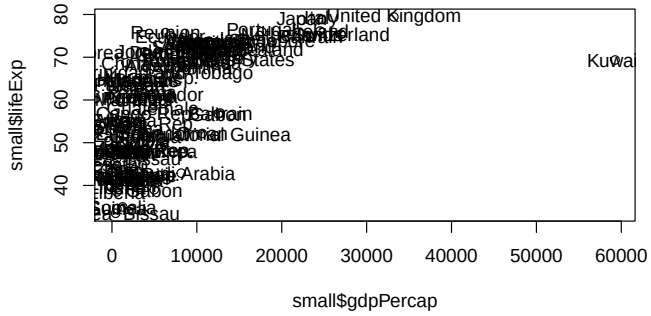


Figure 13.11:

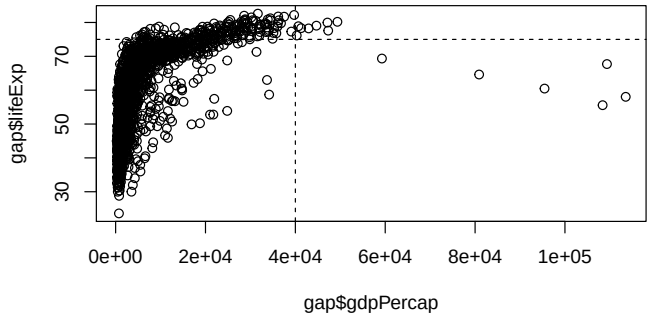


Figure 13.12:

Why ggplot?

- More elegant & compact code than R base graphics
- More aesthetically pleasing defaults than lattice
- Very powerful for exploratory data analysis
- Follows a grammar, just like any language.
- It defines basic components that make up a sentence. In this case, the grammar defines components in a plot.
- *G*rammar of *g*raphics originally coined by Lee Wilkinson

13.3.1 Grammar

The general call for `ggplot2` looks like this:

```
ggplot(data=, aes(x=, y=), color=, size=,) + geom_xxxx()+geom_yyyy()
```

The *grammar* involves some basic components:

1. **Data**: a data.frame
2. **Aes**thetics: How your data are represented visually, aka its "mapping". Which variables are shown on x, y axes, as well as color, size, shape, etc.
3. **Geom**etry: The geometric objects in a plot – points, lines, polygons, etc.

The key to understanding `ggplot2` is thinking about a figure in layers: just like you might do in an image editing program like Photoshop, Illustrator, or Inkscape.

Let's look at an example:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

So the first thing we do is call the `ggplot` function. This function lets R know that we're creating a new plot, and any of the arguments we give the `ggplot` function are the global options for the plot: they apply to all layers on the plot.

We've passed in two arguments to `ggplot`. First, we tell `ggplot` what **`data`** we want to show on our figure, in this example the `gapminder` data we read in earlier.

For the second argument we passed in the **`aes`** function, which tells `ggplot` how variables in the data map to aesthetic properties of the figure, in this case the x and y locations. Here we told `ggplot` we want to plot the `lifeExp` column of the gapminder data frame on the x-axis, and the `gdpPercap` column on the y-axis.

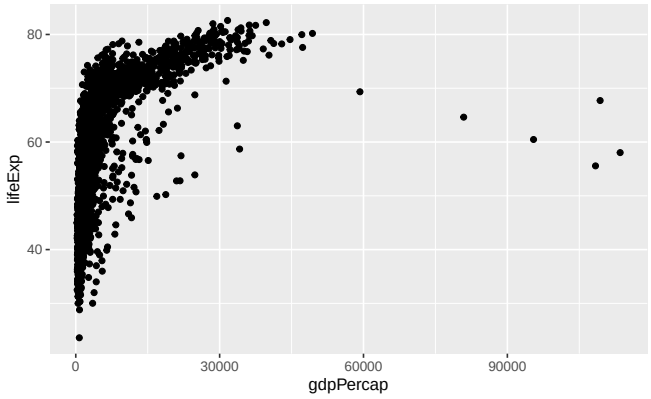
Notice that we didn't need to explicitly pass `aes` these columns (e.g. `x = gapminder[, "lifeExp""])`, this is because `ggplot` is smart enough to know to look in the data for that column!

By itself, the call to `ggplot` isn't enough to draw a figure:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp))
```

We need to tell `ggplot` how we want to visually represent the data, which we do by adding a new **`geom`** layer. In our example, we used `geom_point`, which tells `ggplot` we want to visually represent the relationship between x and y as a scatterplot of points:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()

# same as
# my_plot <- ggplot(data = gap, aes(x = gdpPercap, y = lifeExp))
# my_plot + geom_point()
```



Challenge 1.

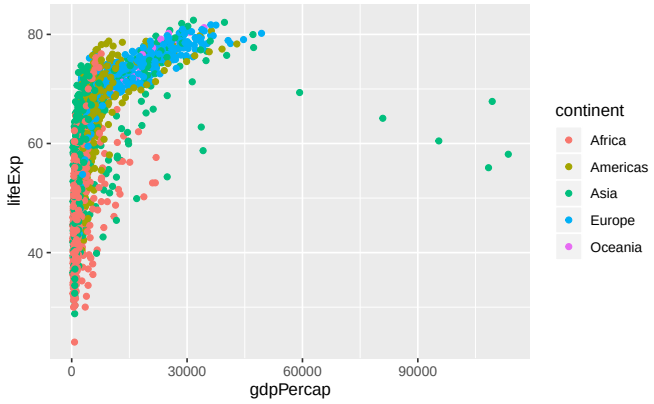
Modify the example so that the figure visualises how life expectancy has changed over time:

Hint: the gapminder dataset has a column called `year`, which should appear on the x-axis.

13.3.2 Anatomy of `aes`

In the previous examples and challenge we've used the **`aes`** function to tell the scatterplot `geom` about the **`x`** and **`y`** locations of each point. Another aesthetic property we can modify is the point **color**.

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point()
```



Normally, specifying options like `color="red"` or `size=10` for a given layer results in its contents being red and quite large. Inside the `aes()` function, however, these arguments are given entire variables whose values will then be displayed using different realizations of that aesthetic.

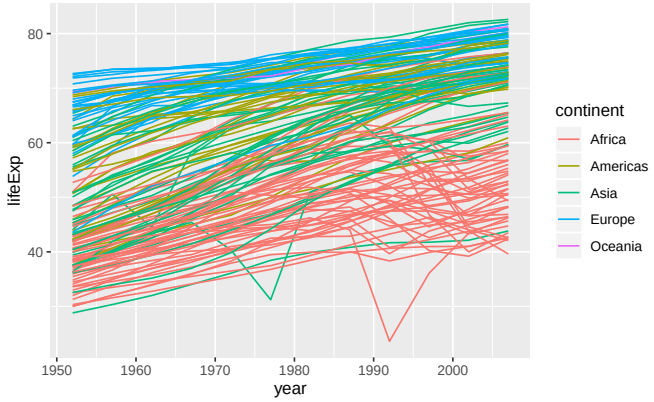
Color isn't the only aesthetic argument we can set to display variation in the data. We can also vary by shape, size, etc.

```
ggplot(data=, aes(x=, y=, by =, color=, linetype=, shape=, size=))
```

13.3.3 Layers

In the previous challenge, you plotted `lifExp` over time. Using a scatterplot probably isn't the best for visualising change over time. Instead, let's tell `ggplot` to visualise the data as a line plot:

```
ggplot(data = gap, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line()
```

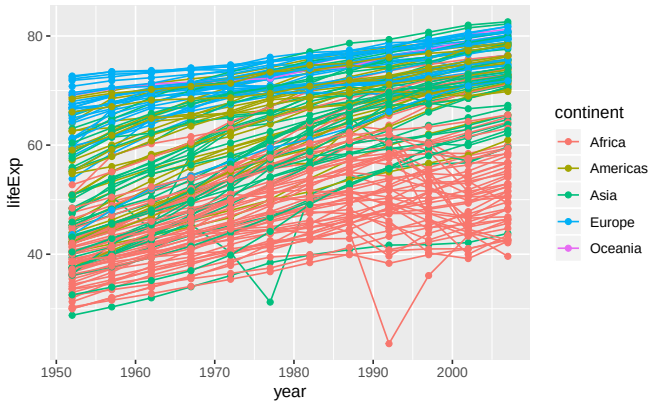


Instead of adding a `geom_point` layer, we've added a `geom_line` layer. We've

also added the **by** aesthetic, which tells `ggplot` to draw a line for each country.

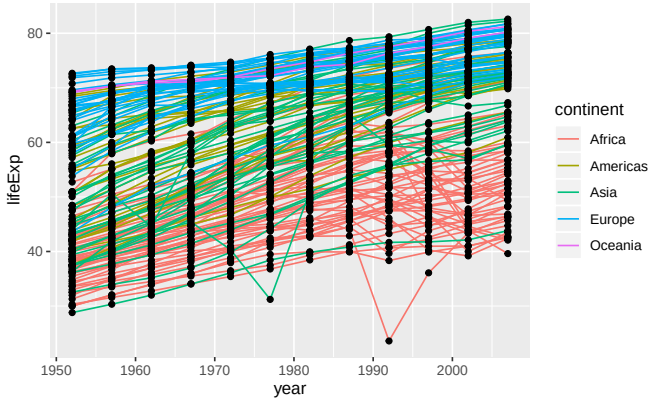
But what if we want to visualise both lines and points on the plot? We can simply add another layer to the plot:

```
ggplot(data = gap, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line() +
  geom_point()
```



It's important to note that each layer is drawn on top of the previous layer. In this example, the points have been drawn on top of the lines. Here's a demonstration:

```
ggplot(data = gap, aes(x=year, y=lifeExp, by=country)) +
  geom_line(aes(color=continent)) +
  geom_point()
```



In this example, the aesthetic mapping of **color** has been moved from the global plot options in `ggplot` to the `geom_line` layer so it no longer applies to the points. Now we can clearly see that the points are drawn on top of the lines.

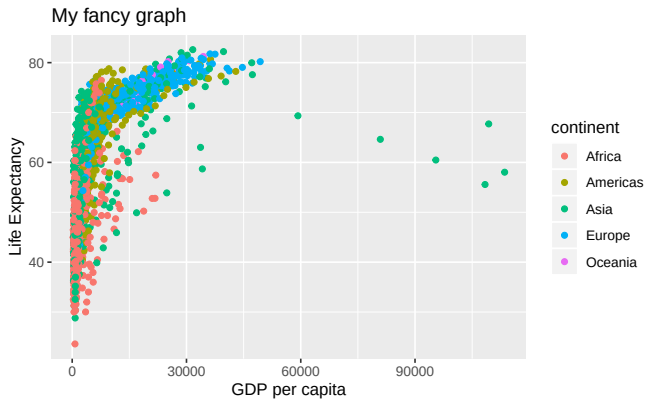
Challenge 2.

Switch the order of the point and line layers from the previous example. What happened?

13.3.4 Labels

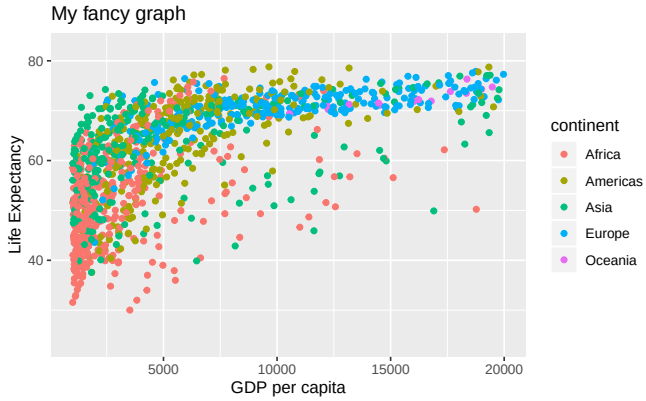
Labels are considered to be their own layers in `ggplot`.

```
# add x and y axis labels
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  xlab("GDP per capita") +
  ylab("Life Expectancy") +
  ggtitle("My fancy graph")
```



So are scales:

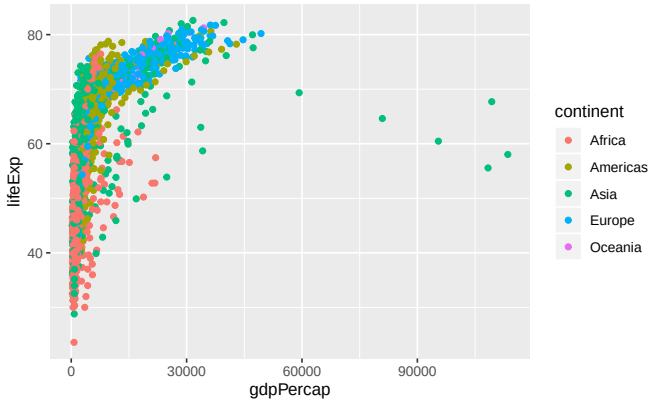
```
# limit x-axis from 1,000 to 20,000
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  xlab("GDP per capita") +
  ylab("Life Expectancy") +
  ggtitle("My fancy graph") +
  xlim(1000, 20000)
#> Warning: Removed 515 rows containing missing values (geom_point).
```



13.3.5 Transformations and Stats

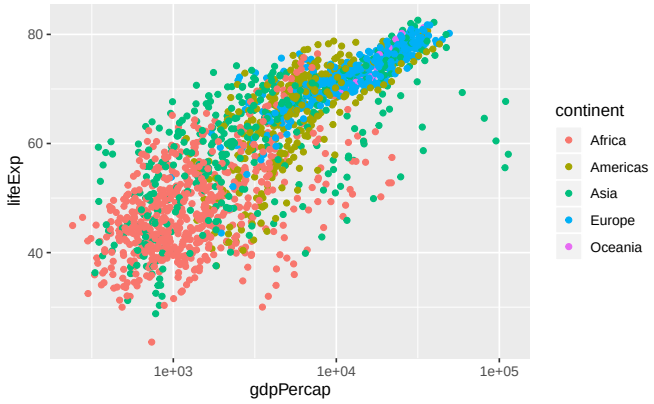
`ggplot` also makes it easy to overlay statistical models over the data. To demonstrate we'll go back to an earlier example:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point()
```



We can change the scale of units on the x-axis using the `scale` functions. These control the mapping between the data values and visual values of an aesthetic.

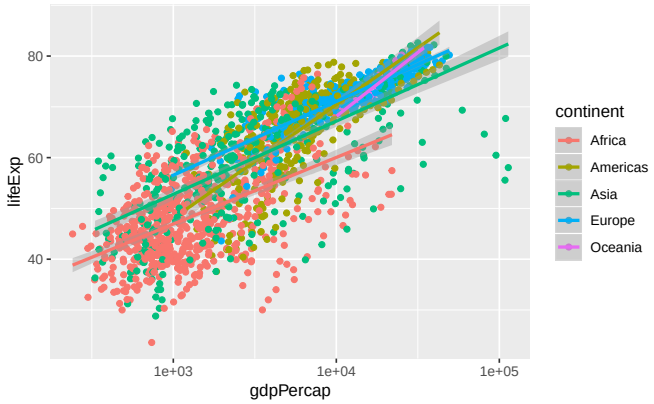
```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  scale_x_log10()
```



The `log10` function applied a transformation to the values of the `gdpPercap` column before rendering them on the plot, so that each multiple of 10 now only corresponds to an increase in 1 on the transformed scale, e.g. a GDP per capita of 1,000 is now 3 on the y axis, a value of 10,000 corresponds to 4 on the x-axis and so on. This makes it easier to visualise the spread of data on the x-axis.

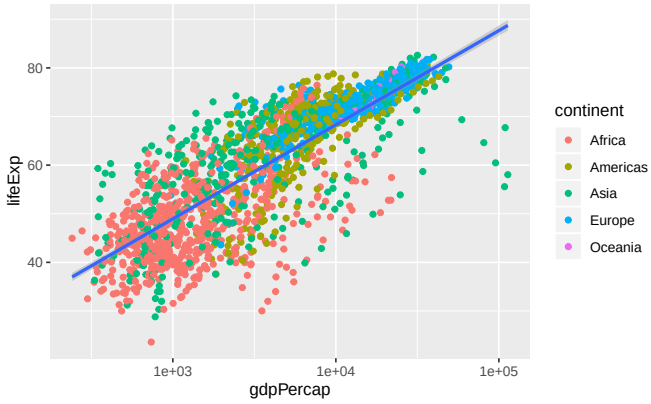
We can fit a simple relationship to the data by adding another layer, `geom_smooth`:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  scale_x_log10() +
  geom_smooth(method="lm")
```



Note that we have 5 lines, one for each region, because of the **color** option is the global `aes` function. But if we move it, we get different results:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(color=continent)) +
  scale_x_log10() +
  geom_smooth(method="lm")
```



So, there are two ways an aesthetic can be specified. Here, we set the **color** aesthetic by passing it as an argument to `geom_point`. Previously in the lesson, we used the `aes` function to define a *mapping* between data variables and their visual representation.

We can make the line thicker by setting the **size** aesthetic in the geom_smooth layer:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(color=continent)) +
  scale_x_log10() +
  geom_smooth(method="lm", size = 1.5)
```

Challenge 3.

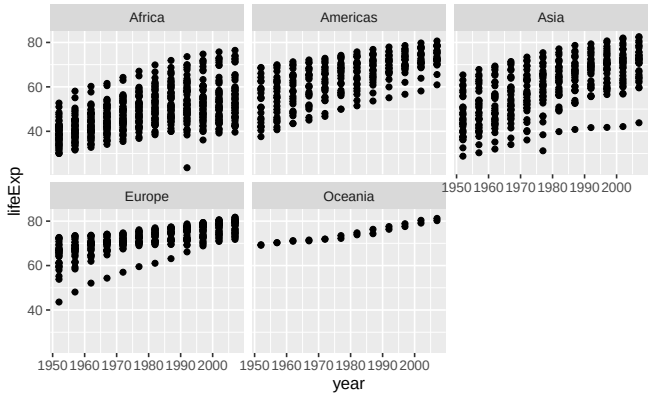
Modify the color and size of the points on the point layer in the previous example so that they are fixed (i.e. not reflective of continent).

Hint: do not use the `aes` function.

13.3.6 Facets

Earlier, we visualised the change in life expectancy over time across all countries in one plot. Alternatively, we can split this out over multiple panels by adding a layer of **facet** panels:

```
ggplot(data = gap, aes(x = year, y = lifeExp)) +
  geom_point() +
  facet_wrap( ~ continent)
```



13.3.7 Putting Everything Together

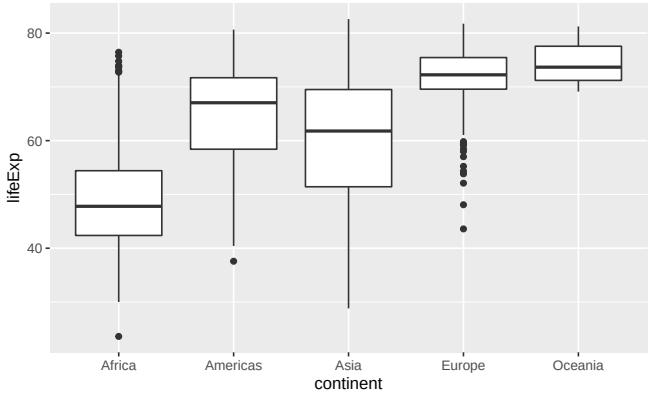
Here are some other common `geom` layers:

bar plots

```
# count of lifeExp bins
ggplot(data = gap, aes(x = lifeExp)) +
  geom_bar(stat="bin")
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

# with color representing regions
ggplot(data = gap, aes(x = lifeExp, fill = continent)) +
  geom_bar(stat="bin")
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

box plots

```
ggplot(data = gap, aes(x = continent, y = lifeExp)) +
  geom_boxplot()
```



This is just a taste of what you can do with `ggplot2`.

RStudio provides a really useful cheat sheet of the different layers available, and more extensive documentation is available on the ggplot2 website.

Finally, if you have no idea how to change something, a quick Google search will usually send you to a relevant question and answer on Stack Overflow with reusable code to modify!

Challenge 4.

Create a density plot of GDP per capita, filled by continent.

Advanced: - Transform the x-axis to better visualise the data spread. - Add a facet layer to panel the density plots by year.

13.4 Saving plots

There are two basic image types:

1) **Raster/Bitmap** (.png, .jpeg)

Every pixel of a plot contains its own separate coding; not so great if you want to resize the image.

```
jpeg(filename="example.png", width=, height=)
plot(x,y)
dev.off()
```

2) **Vector** (.pdf, .ps)

Every element of a plot is encoded with a function that gives its coding conditional on several factors; this is great for resizing.

```
pdf(filename="example.pdf", width=, height=)
plot(x,y)
dev.off()
```

Exporting with ggplot

```
# Assume we saved our plot is an object called example.plot
ggsave(filename="example.pdf", plot=example.plot, scale=, width=, height=)
```

Chapter 14

Statistical Inferences

```
# setup
gap <- read.csv("data/gapminder-FiveYearData.csv", stringsAsFactors = TRUE)
```

14.1 Statistical Distributions

Since R was developed by statisticians, it handles distributions and simulation seamlessly.

All commonly-used distributions have functions in R. Each distribution has a family of functions:

- `d` - probability density/mass function, e.g. `dnorm()`
- `r` - generate a random value, e.g., `rnorm()`
- `p` - cumulative distribution function, e.g., `pnorm()`
- `q` - quantile function (inverse CDF), e.g., `qnorm()`

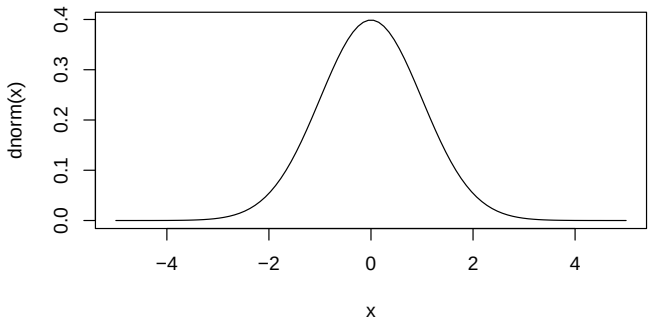
Let's see some of these functions in action with the normal distribution (mean 0, standard deviation 1)

```
dnorm(1.96) # probability density of 1.96 from normal distribution
#> [1] 0.0584
rnorm(1:10) # get 10 random values from the normal distribution
#>  [1] -1.40004  0.25532 -2.43726 -0.00557  0.62155  1.14841 -1.82182 -0.24733
#>  [9] -0.24420 -0.28271
pnorm(1.96) # cumultative distribution function
#> [1] 0.975
qnorm(.975) # inverse cumulative distribution function
#> [1] 1.96
```

We can also use these functions on other distributions:

- `rnorm()` # normal distribution
- `runif()` # uniform distribution
- `rbinom()` # binomial distribution
- `rpois()` # poisson distribtion
- `rbeta()` # beta distribution
- `rgamma()` # gamma distribution
- `rt()` # student t distribution
- `rchisq()`. # chi-squared distribution

```
rbinom(0:10, size = 10, prob = 0.3)
#>  [1] 2 4 4 2 6 4 1 3 4 4 1
dt(5, df = 1)
#> [1] 0.0122

x <- seq(-5, 5, length = 100)
plot(x, dnorm(x), type = 'l')
```



14.1.1 Sampling and Simulation

We can draw a sample with or without replacement with `sample`.

```
sample(1:nrow(gap), 20, replace = FALSE)
#>  [1]  385  513 1084  815  735 1201 1611  307  368 1153  846 1087 1118  163 1294
#> [16] 1300 1673 1638  657  778
```

`dplyr` has a helpful `select_n` function that samples rows of a dataframe.

```
small <- sample_n(gap, 20)
nrow(small)
#> [1] 20
```

Here's an example of some code that would be part of a bootstrap.

```
gap <- read.csv("data/gapminder-FiveYearData.csv", stringsAsFactors = F)
```

```
# actual mean
mean(gap$lifeExp, na.rm = TRUE)
#> [1] 59.5

# here's a bootstrap sample:
smp <- sample_n(gap, size = nrow(gap), replace = TRUE)
mean(smp$lifeExp, na.rm = TRUE)
#> [1] 59.2
```

14.1.2 Random Seeds

A few key facts about generating random numbers:

- Random numbers on a computer are *pseudo-random*; they are generated deterministically from a very, very, very long sequence that repeats
- The *seed* determines where you are in that sequence

To replicate any work involving random numbers, make sure to set the seed first. The seed can be arbitrary – pick your favorite number.

```
set.seed(1)
vals <- sample(1:nrow(gap), 10)
vals
#>  [1]  453  634  975 1545  343 1527 1605 1122 1067  105

vals <- sample(1:nrow(gap), 10)
vals
#>  [1]  351  301 1170  654 1309  846 1219 1684  645 1318

set.seed(1)
vals <- sample(1:nrow(gap), 10)
vals
#>  [1]  453  634  975 1545  343 1527 1605 1122 1067  105
```

14.1.3 Challenges

Challenge 1.

Generate 100 random Poisson values with a population mean of 5. How close is the mean of those 100 values to the value of 5?

Challenge 2.

What is the 95th percentile of a chi-square distribution with 1 degree of freedom?

Challenge 3.

What's the probability of getting a value greater than 5 if you draw from a standard normal distribution? What about a t distribution with 1 degree of freedom?

14.2 Inferences and Regressions

Once we've imported our data, summarized it, carried out group-wise operations, and perhaps reshaped it, we may also want to attempt causal inference.

This often requires doing the following:

1) Carrying out Classical Hypothesis Tests
2) Estimating Regressions

```
# setup
gap <- read.csv("data/gapminder-FiveYearData.csv", stringsAsFactors = F)
```

14.2.1 Statistical Tests

Let's say we're interested in whether the life expectancy in 1967 is different than in 1977.

```
# pull out life expectancy by different years
life.exp.1967 <- gap$lifeExp[gap$year==1967]
life.exp.1977 <- gap$lifeExp[gap$year==1977]
```

One can test for differences in distributions in either:

1) Their means using t-tests:

```
# t test of means
t.test(x = life.exp.1967, y = life.exp.1977)
#>
#>  Welch Two Sample t-test
#>
#> data:  life.exp.1967 and life.exp.1977
#> t = -3, df = 281, p-value = 0.005
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -6.57 -1.21
#> sample estimates:
#> mean of x mean of y
#>      55.7      59.6
```

2) Their entire distributions using ks-tests

```
# ks tests of distributions
ks.test(x = life.exp.1967, y = life.exp.1977)
#> Warning in ks.test(x = life.exp.1967, y = life.exp.1977): p-value will be
#> approximate in the presence of ties
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#> data:  life.exp.1967 and life.exp.1977
#> D = 0.2, p-value = 0.008
#> alternative hypothesis: two-sided
```

14.2.2 Regressions and Linear Models

Running regressions in R is generally straightforward. There are two basic, catch-all regression functions in R:

- *glm* fits a generalized linear model with your choice of family/link function (gaussian, logit, poisson, etc.)
- *lm* is just a standard linear regression (equivalent to glm with family = gaussian(link = "identity"))

The basic `glm` call looks something like this:

```
glm(formula = y ~ x1 + x2 + x3 + ..., family = familyname(link = "linkname"), data = )
```

There are a bunch of families and links to use (`?family` for a full list), but some essentials are: **binomial(link = "logit")**, **gaussian(link = "identity")**, and **poisson(link = "log")**

If you're using `lm`, the call looks the same but without the `family` argument.

- Example: suppose we want to regress the life expectency on the GDP per capita and the population, as well as the continent and year. The `lm` call would be something like this:

```
reg <- lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent + year, data = gap)
```

Missing values

Missing values obviously cannot convey any information about the relationship between the variables. Most modeling functions will drop any rows that contain missing values.

14.2.3 Regression Output

When we store this regression in an object, we get access to several items of interest.

1. All components contained in the regression output:

```
names(reg)
#>  [1] "coefficients"  "residuals"     "effects"       "rank"
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"
#>  [9] "contrasts"     "xlevels"       "call"          "terms"
#> [13] "model"
```

2. Regression coefficients

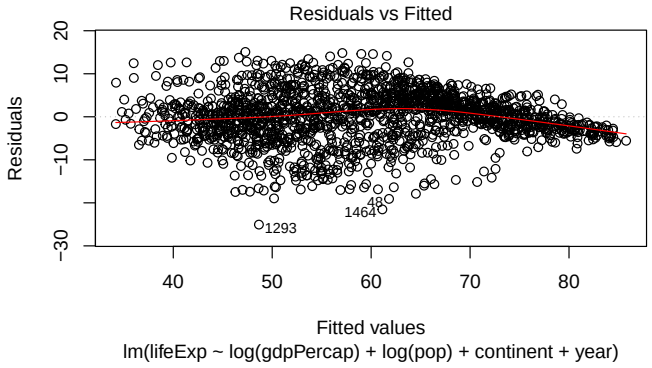
```
reg$coefficients
#>       (Intercept)    log(gdpPercap)          log(pop) continentAmericas
#>          -460.813             5.076             0.153             8.745
#>     continentAsia   continentEurope  continentOceania              year
#>             6.825            12.281            12.540             0.238
```

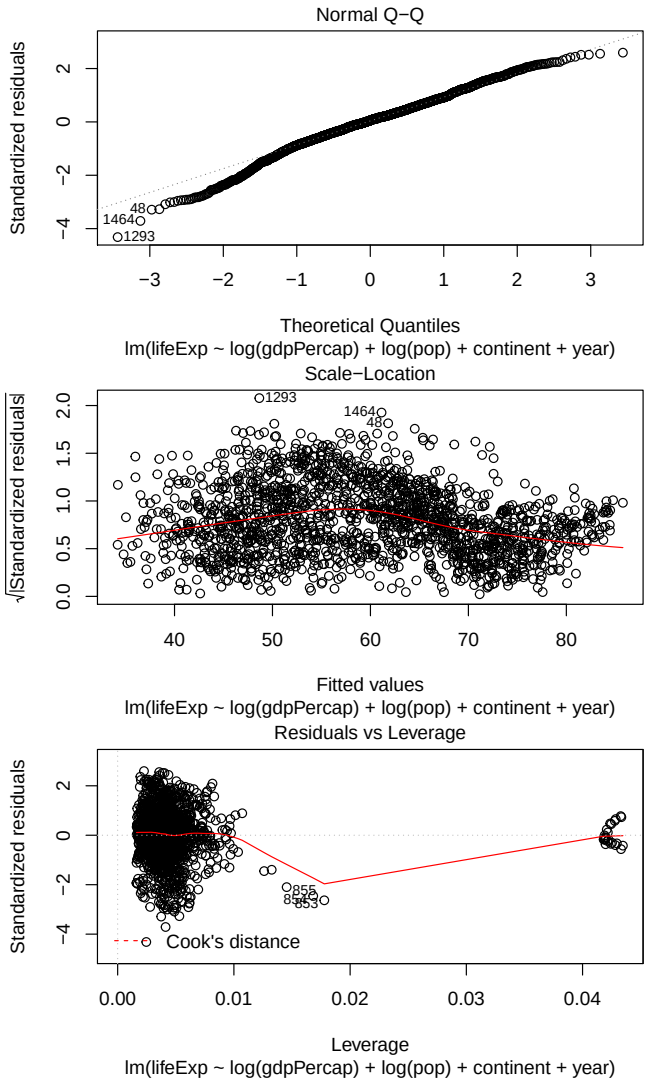
3. Regression degrees of freedom

```
reg$df.residual
#> [1] 1696
```

4. Standard (diagnostic) plots for a regression

```
plot(reg)
```





R also has a helpful `summary` method for regression objects.

```
summary(reg)
#>
#> Call:
#> lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent +
#>     year, data = gap)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -25.057  -3.286   0.329   3.706  15.065
#>
```

```
#> Coefficients:
#>                    Estimate Std. Error t value Pr(>|t|)
#> (Intercept)       -4.61e+02   1.70e+01  -27.15   <2e-16 ***
#> log(gdpPercap)     5.08e+00   1.63e-01   31.19   <2e-16 ***
#> log(pop)           1.53e-01   9.67e-02    1.58     0.11
#> continentAmericas  8.75e+00   4.77e-01   18.35   <2e-16 ***
#> continentAsia      6.83e+00   4.23e-01   16.13   <2e-16 ***
#> continentEurope    1.23e+01   5.29e-01   23.20   <2e-16 ***
#> continentOceania   1.25e+01   1.28e+00    9.79   <2e-16 ***
#> year               2.38e-01   8.93e-03   26.61   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 5.81 on 1696 degrees of freedom
#> Multiple R-squared:  0.798,  Adjusted R-squared:  0.798
#> F-statistic:  960 on 7 and 1696 DF,  p-value: <2e-16
```

We can also extract useful things from the summary object:

```
# Store summary method results
summ_reg <- summary(reg)

# View summary method results objects
objects(summ_reg)
#>  [1] "adj.r.squared" "aliased"       "call"          "coefficients"
#>  [5] "cov.unscaled"  "df"            "fstatistic"    "r.squared"
#>  [9] "residuals"     "sigma"         "terms"

# View table of coefficients
summ_reg$coefficients
#>                   Estimate Std. Error t value  Pr(>|t|)
#> (Intercept)       -460.813   16.97028  -27.15 3.96e-135
#> log(gdpPercap)       5.076    0.16272   31.19 3.37e-169
#> log(pop)             0.153    0.09668    1.58  1.14e-01
#> continentAmericas    8.745    0.47660   18.35  9.61e-69
#> continentAsia        6.825    0.42320   16.13  1.49e-54
#> continentEurope     12.281    0.52924   23.20 1.12e-103
#> continentOceania    12.540    1.28114    9.79  4.80e-22
#> year                 0.238    0.00893   26.61 1.06e-130
```

14.2.4 Interactions

There are also some useful shortcuts for regressing on interaction terms:

1. `x1:x2` interacts all terms in x1 with all terms in x2

```
mod.1 <- lm(lifeExp ~ log(gdpPercap) + log(pop) + continent:factor(year), data = gap)
summary(mod.1)
#>
#> Call:
#> lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent:factor(year),
#>     data = gap)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -26.568  -2.553   0.004   2.915  15.567
#>
#> Coefficients: (1 not defined because of singularities)
#>                                     Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                          27.1838     4.6849    5.80  7.8e-09 ***
#> log(gdpPercap)                        5.0795     0.1605   31.65  < 2e-16 ***
#> log(pop)                              0.0789     0.0943    0.84  0.40251
#> continentAfrica:factor(year)1952    -24.1425     4.1125   -5.87  5.2e-09 ***
#> continentAmericas:factor(year)1952  -16.4465     4.1663   -3.95  8.2e-05 ***
#> continentAsia:factor(year)1952      -19.3347     4.1408   -4.67  3.3e-06 ***
#> continentEurope:factor(year)1952     -7.0918     4.1352   -1.71  0.08654 .
#> continentOceania:factor(year)1952    -6.0635     5.6511   -1.07  0.28344
#> continentAfrica:factor(year)1957    -22.4964     4.1098   -5.47  5.1e-08 ***
#> continentAmericas:factor(year)1957  -14.3673     4.1643   -3.45  0.00057 ***
#> continentAsia:factor(year)1957      -17.1743     4.1375   -4.15  3.5e-05 ***
#> continentEurope:factor(year)1957     -5.9094     4.1327   -1.43  0.15293
#> continentOceania:factor(year)1957    -5.6300     5.6503   -1.00  0.31921
#> continentAfrica:factor(year)1962    -21.0139     4.1069   -5.12  3.5e-07 ***
#> continentAmericas:factor(year)1962  -12.3135     4.1630   -2.96  0.00314 **
#> continentAsia:factor(year)1962      -15.5626     4.1351   -3.76  0.00017 ***
#> continentEurope:factor(year)1962     -5.0542     4.1308   -1.22  0.22130
#> continentOceania:factor(year)1962    -5.3122     5.6498   -0.94  0.34723
#> continentAfrica:factor(year)1967    -19.7034     4.1035   -4.80  1.7e-06 ***
#> continentAmericas:factor(year)1967  -10.9324     4.1613   -2.63  0.00869 **
#> continentAsia:factor(year)1967      -13.1569     4.1327   -3.18  0.00148 **
#> continentEurope:factor(year)1967     -4.9134     4.1291   -1.19  0.23423
#> continentOceania:factor(year)1967    -5.7712     5.6492   -1.02  0.30712
#> continentAfrica:factor(year)1972    -18.1469     4.1007   -4.43  1.0e-05 ***
#> continentAmericas:factor(year)1972   -9.6537     4.1595   -2.32  0.02042 *
#> continentAsia:factor(year)1972      -11.6014     4.1293   -2.81  0.00502 **
#> continentEurope:factor(year)1972     -4.9763     4.1275   -1.21  0.22813
#> continentOceania:factor(year)1972    -5.8094     5.6487   -1.03  0.30389
#> continentAfrica:factor(year)1977    -16.1848     4.0996   -3.95  8.2e-05 ***
#> continentAmericas:factor(year)1977   -8.3382     4.1580   -2.01  0.04509 *
#> continentAsia:factor(year)1977      -10.1220     4.1270   -2.45  0.01428 *
#> continentEurope:factor(year)1977     -4.5523     4.1267   -1.10  0.27013
```

```
#> continentOceania:factor(year)1977   -5.1232     5.6485   -0.91  0.36454
#> continentAfrica:factor(year)1982   -14.1933     4.0990   -3.46  0.00055 ***
#> continentAmericas:factor(year)1982  -6.5921     4.1577   -1.59  0.11304
#> continentAsia:factor(year)1982      -7.6001     4.1257   -1.84  0.06564 .
#> continentEurope:factor(year)1982    -4.1185     4.1262   -1.00  0.31837
#> continentOceania:factor(year)1982   -4.0553     5.6483   -0.72  0.47288
#> continentAfrica:factor(year)1987   -12.1850     4.0995   -2.97  0.00300 **
#> continentAmericas:factor(year)1987  -4.7157     4.1577   -1.13  0.25687
#> continentAsia:factor(year)1987      -5.6914     4.1249   -1.38  0.16785
#> continentEurope:factor(year)1987    -3.7298     4.1258   -0.90  0.36613
#> continentOceania:factor(year)1987   -3.5164     5.6480   -0.62  0.53364
#> continentAfrica:factor(year)1992   -11.8028     4.0994   -2.88  0.00404 **
#> continentAmericas:factor(year)1992  -3.2855     4.1575   -0.79  0.42949
#> continentAsia:factor(year)1992      -4.3823     4.1241   -1.06  0.28811
#> continentEurope:factor(year)1992    -2.5151     4.1262   -0.61  0.54225
#> continentOceania:factor(year)1992   -1.9804     5.6480   -0.35  0.72590
#> continentAfrica:factor(year)1997   -11.9577     4.0986   -2.92  0.00358 **
#> continentAmericas:factor(year)1997  -2.1611     4.1566   -0.52  0.60319
#> continentAsia:factor(year)1997      -3.5016     4.1228   -0.85  0.39583
#> continentEurope:factor(year)1997    -2.0843     4.1256   -0.51  0.61348
#> continentOceania:factor(year)1997   -1.4478     5.6478   -0.26  0.79771
#> continentAfrica:factor(year)2002   -12.5237     4.0972   -3.06  0.00227 **
#> continentAmericas:factor(year)2002  -0.9898     4.1564   -0.24  0.81180
#> continentAsia:factor(year)2002      -2.6798     4.1221   -0.65  0.51571
#> continentEurope:factor(year)2002    -1.5734     4.1252   -0.38  0.70294
#> continentOceania:factor(year)2002   -0.4735     5.6477   -0.08  0.93320
#> continentAfrica:factor(year)2007   -11.6568     4.0948   -2.85  0.00447 **
#> continentAmericas:factor(year)2007  -0.6931     4.1550   -0.17  0.86754
#> continentAsia:factor(year)2007      -2.2008     4.1202   -0.53  0.59332
#> continentEurope:factor(year)2007    -1.5284     4.1247   -0.37  0.71102
#> continentOceania:factor(year)2007        NA         NA      NA       NA
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 5.65 on 1642 degrees of freedom
#> Multiple R-squared:  0.816,  Adjusted R-squared:  0.809
#> F-statistic:  119 on 61 and 1642 DF,  p-value: <2e-16
```

2. `x1*x2` produces the cross of x1 and x2, or x1+x2+x1:x2

```
mod.2 <- lm(lifeExp ~ log(gdpPercap) + log(pop) + continent*factor(year), data = gap)
summary(mod.2)
#>
#> Call:
#> lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent *
#>     factor(year), data = gap)
```

```
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -26.568  -2.553   0.004   2.915  15.567
#>
#> Coefficients:
#>                                    Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                          3.0413     2.0741    1.47  0.14275
#> log(gdpPercap)                       5.0795     0.1605   31.65  < 2e-16 ***
#> log(pop)                             0.0789     0.0943    0.84  0.40251
#> continentAmericas                    7.6960     1.3932    5.52  3.9e-08 ***
#> continentAsia                        4.8078     1.2657    3.80  0.00015 ***
#> continentEurope                     17.0508     1.3295   12.83  < 2e-16 ***
#> continentOceania                    18.0790     4.0890    4.42  1.0e-05 ***
#> factor(year)1957                     1.6461     1.1078    1.49  0.13747
#> factor(year)1962                     3.1286     1.1084    2.82  0.00482 **
#> factor(year)1967                     4.4392     1.1097    4.00  6.6e-05 ***
#> factor(year)1972                     5.9956     1.1113    5.39  7.9e-08 ***
#> factor(year)1977                     7.9578     1.1124    7.15  1.3e-12 ***
#> factor(year)1982                     9.9492     1.1134    8.94  < 2e-16 ***
#> factor(year)1987                    11.9575     1.1138   10.74  < 2e-16 ***
#> factor(year)1992                    12.3398     1.1146   11.07  < 2e-16 ***
#> factor(year)1997                    12.1848     1.1161   10.92  < 2e-16 ***
#> factor(year)2002                    11.6188     1.1181   10.39  < 2e-16 ***
#> factor(year)2007                    12.4857     1.1212   11.14  < 2e-16 ***
#> continentAmericas:factor(year)1957   0.4330     1.9438    0.22  0.82375
#> continentAsia:factor(year)1957       0.5142     1.7776    0.29  0.77241
#> continentEurope:factor(year)1957    -0.4638     1.8313   -0.25  0.80010
#> continentOceania:factor(year)1957   -1.2126     5.7552   -0.21  0.83315
#> continentAmericas:factor(year)1962   1.0043     1.9438    0.52  0.60546
#> continentAsia:factor(year)1962       0.6435     1.7777    0.36  0.71741
#> continentEurope:factor(year)1962    -1.0911     1.8315   -0.60  0.55142
#> continentOceania:factor(year)1962   -2.3774     5.7552   -0.41  0.67960
#> continentAmericas:factor(year)1967   1.0750     1.9438    0.55  0.58033
#> continentAsia:factor(year)1967       1.7387     1.7777    0.98  0.32819
#> continentEurope:factor(year)1967    -2.2608     1.8317   -1.23  0.21728
#> continentOceania:factor(year)1967   -4.1468     5.7552   -0.72  0.47130
#> continentAmericas:factor(year)1972   0.7972     1.9438    0.41  0.68176
#> continentAsia:factor(year)1972       1.7377     1.7779    0.98  0.32851
#> continentEurope:factor(year)1972    -3.8801     1.8322   -2.12  0.03435 *
#> continentOceania:factor(year)1972   -5.7415     5.7552   -1.00  0.31862
#> continentAmericas:factor(year)1977   0.1505     1.9439    0.08  0.93828
#> continentAsia:factor(year)1977       1.2549     1.7784    0.71  0.48050
#> continentEurope:factor(year)1977    -5.4183     1.8329   -2.96  0.00316 **
#> continentOceania:factor(year)1977   -7.0175     5.7553   -1.22  0.22290
```

```
#> continentAmericas:factor(year)1982  -0.0948     1.9439   -0.05  0.96110
#> continentAsia:factor(year)1982       1.7854     1.7788    1.00  0.31567
#> continentEurope:factor(year)1982    -6.9759     1.8336   -3.80  0.00015 ***
#> continentOceania:factor(year)1982   -7.9409     5.7553   -1.38  0.16785
#> continentAmericas:factor(year)1987  -0.2267     1.9440   -0.12  0.90720
#> continentAsia:factor(year)1987       1.6858     1.7796    0.95  0.34363
#> continentEurope:factor(year)1987    -8.5955     1.8350   -4.68  3.0e-06 ***
#> continentOceania:factor(year)1987   -9.4104     5.7554   -1.64  0.10223
#> continentAmericas:factor(year)1992   0.8213     1.9441    0.42  0.67276
#> continentAsia:factor(year)1992       2.6127     1.7803    1.47  0.14243
#> continentEurope:factor(year)1992    -7.7631     1.8346   -4.23  2.5e-05 ***
#> continentOceania:factor(year)1992   -8.2567     5.7555   -1.43  0.15160
#> continentAmericas:factor(year)1997   2.1006     1.9443    1.08  0.28012
#> continentAsia:factor(year)1997       3.6483     1.7812    2.05  0.04070 *
#> continentEurope:factor(year)1997    -7.1773     1.8358   -3.91  9.6e-05 ***
#> continentOceania:factor(year)1997   -7.5691     5.7557   -1.32  0.18867
#> continentAmericas:factor(year)2002   3.8379     1.9442    1.97  0.04854 *
#> continentAsia:factor(year)2002       5.0361     1.7814    2.83  0.00476 **
#> continentEurope:factor(year)2002    -6.1005     1.8369   -3.32  0.00092 ***
#> continentOceania:factor(year)2002   -6.0287     5.7558   -1.05  0.29506
#> continentAmericas:factor(year)2007   3.2677     1.9444    1.68  0.09303 .
#> continentAsia:factor(year)2007       4.6483     1.7823    2.61  0.00919 **
#> continentEurope:factor(year)2007    -6.9223     1.8378   -3.77  0.00017 ***
#> continentOceania:factor(year)2007   -6.4222     5.7558   -1.12  0.26468
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 5.65 on 1642 degrees of freedom
#> Multiple R-squared:  0.816,  Adjusted R-squared:  0.809
#> F-statistic:  119 on 61 and 1642 DF,  p-value: <2e-16
```

Note that we wrapped the `year` variables into a `factor()` function. By default, R breaks up our variables into their different factor levels (as it will do whenever your regressors have factor levels).

If your data aren't factorized, you can tell `lm/glm` to factorize a variable (i.e. create dummy variables on the fly) by writing `factor()`

```
glm(formula = y ~ x1 + x2 + factor(x3), family = family(link = "link"),
            data = )
```

14.2.5 Formatting Regression Tables

Most papers report the results of regression analysis in some kind of table. Typically, this table includes the values of coefficients, standard errors, and

significance levels from one or more models.

The `stargazer` package provides excellent tools to make and format regression tables automatically. It can also output summary statistics from a dataframe:

```
library(stargazer)
stargazer(gap, type = "text")
#>
#> ================================================================================================
#> Statistic   N        Mean          St. Dev.        Min    Pctl(25)     Pctl(75)         Max
#> ------------------------------------------------------------------------------------------------
#> year      1,704   1,980.000        17.300        1,952   1,966.0     1,993.0         2,007
#> pop       1,704 29,601,212.000 106,157,897.000 60,011  2,793,664 19,585,222.0 1,318,683,096
#> lifeExp   1,704     59.500         12.900       23.600   48.200       70.800        82.600
#> gdpPercap 1,704   7,215.000       9,857.000     241.000 1,202.000  9,325.000     113,523.000
#> ------------------------------------------------------------------------------------------------
```

Let's say we want to report the results from three different models:

```
mod.1 <- lm(lifeExp ~ log(gdpPercap) + log(pop), data = gap)
mod.2 <- lm(lifeExp ~ log(gdpPercap) + log(pop) + continent, data = gap)
mod.3 <- lm(lifeExp ~ log(gdpPercap) + log(pop) + continent + year, data = gap)
```

`stargazer` can produce well-formatted tables that hold regression analysis results from all these models side-by-side.

```
stargazer(mod.1, mod.2, mod.3, title = "Regression Results", type = "text")
#>
#> Regression Results
#> ================================================================================================
#>                                                Dependent variable:
#>                          ---------------------------------------------------------------------
#>                                                       lifeExp
#>                                (1)                      (2)                      (3)
#> ------------------------------------------------------------------------------------------------
#> log(gdpPercap)               8.340***                 6.590***                 5.080***
#>                               (0.143)                  (0.182)                  (0.163)
#>
#> log(pop)                     1.280***                 0.866***                  0.153
#>                               (0.111)                  (0.111)                  (0.097)
#>
#> continentAmericas                                     6.170***                 8.740***
#>                                                        (0.555)                  (0.477)
#>
#> continentAsia                                         4.670***                 6.830***
#>                                                        (0.494)                  (0.423)
#>
#> continentEurope                                       8.560***                 12.300***
```

```
#>                                                                (0.608)
#>
#> continentOceania                                             8.350***
#>                                                               (1.510)
#>
#> year
#>
#>
#> Constant                      -28.800***                    -12.000***
#>                                 (2.080)                       (2.270)
#>
#> ------------------------------------------------------------------------
#> Observations                     1,704                         1,704
#> R2                               0.677                         0.714
#> Adjusted R2                      0.677                         0.713
#> Residual Std. Error       7.340 (df = 1701)             6.920 (df = 1697)
#> F Statistic          1,786.000*** (df = 2; 1701)  707.000*** (df = 6; 1697)
#> ========================================================================
#> Note:                                                *p<0.1; **p
```

Customization

`stargazer` is incredibly customizable. Let's say we wanted to:

- re-name our explanatory variables;
- remove information on the "Constant";

- only keep the number of observations from the summary statistics; and
- style the table to look like those in American Journal of Political Science.

```
stargazer(mod.1, mod.2, mod.3, title = "Regression Results", type = "text",
          covariate.labels  = c("GDP per capita, logged", "Population, logged", "Americ
          omit = "Constant",
          keep.stat="n", style = "ajps")
#>
#> Regression Results
#> ------------------------------------------------
#>                                  lifeExp
#>                         Model 1  Model 2   Model 3
#> ------------------------------------------------
#> GDP per capita, logged 8.340*** 6.590*** 5.080***
#>                         (0.143)  (0.182)   (0.163)
#> Population, logged      1.280*** 0.866***   0.153
#>                         (0.111)  (0.111)   (0.097)
#> Americas                         6.170*** 8.740***
```

```
#>                                    (0.555)   (0.477)
#> Asia                              4.670*** 6.830***
#>                                    (0.494)   (0.423)
#> Europe                            8.560*** 12.300***
#>                                    (0.608)   (0.529)
#> Oceania                           8.350*** 12.500***
#>                                    (1.510)   (1.280)
#> Year                                       0.238***
#>                                            (0.009)
#> N                          1704     1704      1704
#> ------------------------------------------------------
#> ***p < .01; **p < .05; *p < .1
```

Check out `?stargazer` to see more options.

Output Types

Once we like the look of our table, we can output/export it in a number of ways. The `type` argument specifies what output the command should produce. Possible values are:

- `"latex"` for LaTeX code,
- `"html"` for HTML code,
- `"text"` for ASCII text output (what we used above).

Let's say we're using LaTeX to typeset our paper. We can output our regression table in LaTeX:

```
stargazer(mod.1, mod.2, mod.3, title = "Regression Results", type = "latex",
          covariate.labels  = c("GDP per capita, logged", "Population, logged", "Americas", "Asia
          omit = "Constant",
          keep.stat="n", style = "ajps")
#>
#> % Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fa
#> % Date and time: Sun, Aug 30, 2020 - 14:59:27
#> \begin{table}[!htbp] \centering
#>   \caption{Regression Results}
#>   \label{}
#> \begin{tabular}{@{\extracolsep{5pt}}lccc}
#> \\[-1.8ex]\hline \\[-1.8ex]
#> \\[-1.8ex] & \multicolumn{3}{c}{\textbf{lifeExp}} \\
#> \\[-1.8ex] & \textbf{Model 1} & \textbf{Model 2} & \textbf{Model 3}\\
#> \hline \\[-1.8ex]
#>  GDP per capita, logged & 8.340$^{***}$ & 6.590$^{***}$ & 5.080$^{***}$ \\
#>   & (0.143) & (0.182) & (0.163) \\
#>   Population, logged & 1.280$^{***}$ & 0.866$^{***}$ & 0.153 \\
```

```
#>   & (0.111) & (0.111) & (0.097) \\
#>   Americas &  & 6.170$^{***}$ & 8.740$^{***}$ \\
#>   &  & (0.555) & (0.477) \\
#>   Asia &  & 4.670$^{***}$ & 6.830$^{***}$ \\
#>   &  & (0.494) & (0.423) \\
#>   Europe &  & 8.560$^{***}$ & 12.300$^{***}$ \\
#>   &  & (0.608) & (0.529) \\
#>   Oceania &  & 8.350$^{***}$ & 12.500$^{***}$ \\
#>   &  & (1.510) & (1.280) \\
#>   Year &  &  & 0.238$^{***}$ \\
#>   &  &  & (0.009) \\
#>  N & 1704 & 1704 & 1704 \\
#> \hline \\[-1.8ex]
#> \multicolumn{4}{l}{$^{***}$p $<$ .01; $^{**}$p $<$ .05; $^{*}$p $<$ .1} \\
#> \end{tabular}
#> \end{table}
```

To include the produced tables in our paper, we can simply insert this stargazer LaTeX output into the publication's TeX source.

Alternatively, you can use the `out` argument to save the output in a .tex or .txt file:

```
stargazer(mod.1, mod.2, mod.3, title = "Regression Results", type = "latex",
          covariate.labels  = c("GDP per capita, logged", "Population, logged", "Americ
          omit = "Constant",
          keep.stat="n", style = "ajps",
          out = "regression-table.txt")
```

To include stargazer tables in Microsoft Word documents (e.g., .doc or .docx), use the following procedure:

- Use the `out` argument to save output into an `.html` file.
- Open the resulting file in your web browser.
- Copy and paste the table from the web browser to your Microsoft Word document.

```
stargazer(mod.1, mod.2, mod.3, title = "Regression Results", type = "html",
          covariate.labels  = c("GDP per capita, logged", "Population, logged", "Americ
          omit = "Constant",
          keep.stat="n", style = "ajps",
          out = "regression-table.html")
```

14.2.6 Challenges

Challenge 1.

Fit two linear regression models from the gapminder data, where the outcome is `lifeExp` and the explanatory variables are `log(pop)`, `log(gdpPercap)`, and `year`. In one model, treat `year` as a numeric variable. In the other, factorize the `year` variable. How do you interpret each model?

Challenge 2.

Fit a logistic regression model where the outcome is whether `lifeExp` is greater than or less than 60 years, exploring the use of different predictors.

Challenge 3.

Using `stargazer`, format a table reporting the results from the three models you created above (two linear regressions and one logistic).

Chapter 15

Strings and Regular Expressions

This unit focuses on chracter (or "string") data. We'll explore:

1. **string basics**, like concatinating and subsettings.
2. **regular expressions**, a powerful cross-language tool for working with string data.
3. **common tools**, that take regex and apply them to real problems.

This chapter will focus on the `stringr` package for string manipulation. `stringr` is not part of the core `tidyverse` because you don't always have textual data, so we need to load it explicitly.

```
library(tidyverse)
library(stringr)
```

15.1 String Basics

15.1.1 Creating Strings

You can create strings with either single quotes or double quotes. Unlike other languages, there is no difference in behavior. I recommend always using `"`, unless you want to create a string that contains multiple `"`.

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'
```

15.1.2 Escape and Special Characters

Single and double quotes are known as "metacharacters," meaning that they have special meaning to the R language. To include a literal single or double quote in a string you can use \ to "escape" it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

That means if you want to include a literal backslash, you'll need to double it up: `"\\"`.

Beware that the printed representation of a string is not the same as string itself, because the printed representation shows the escapes. To see the raw contents of the string, use `writeLines()`:

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

There are a handful of other special characters. The most common are `"\n"`, newline, and `"\t"`, tab, but you can see the complete list by requesting help on `"`: `?'"'`, or `?"'"`. You'll also sometimes see strings like `"\u00b5"`, this is a way of writing non-English characters that works on all platforms:

```
x <- "\u00b5"
x
#> [1] "µ"
```

Multiple strings are often stored in a character vector, which you can create with `c()`:

```
c("one", "two", "three")
#> [1] "one"   "two"   "three"
```

15.1.3 String length

Base R contains many functions to work with strings but we'll avoid them because they can be inconsistent, which makes them hard to remember. Instead we'll use functions from `stringr`. These have more intuitive names, and all start with `str_`. For example, `str_length()` tells you the number of characters in a string:

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

The common `str_` prefix is particularly useful if you use RStudio, because typing `str_` will trigger autocomplete, allowing you to see all `stringr` functions:

15.1.4 Combining strings

To combine two or more strings, use `str_c()`:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Use the `sep` argument to control how they're separated:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

`str_c()` is vectorised, and it automatically recycles shorter vectors to the same length as the longest:

```
x <- c("a", "b", "c")
str_c("prefix-", x)
#> [1] "prefix-a" "prefix-b" "prefix-c"
```

To collapse a vector of strings into a single string, use `collapse`:

```
x <- c("x", "y", "z")
str_c(x, collapse = ", ")
#> [1] "x, y, z"
```

15.1.5 Subsetting strings

You can extract parts of a string using `str_sub()`. As well as the string, `str_sub()` takes `start` and `end` arguments, which give the (inclusive) position of the substring:

```
x <- c("Rochelle is the Greatest")
str_sub(x, 1, 8)
#> [1] "Rochelle"

# negative numbers count backwards from end
str_sub(x, -8, -1)
#> [1] "Greatest"
```

Note that `str_sub()` won't fail if the string is too short: it will just return as much as possible:

```
str_sub("a", 1, 3)
#> [1] "a"
```

You can also use the assignment form of `str_sub()` to modify strings:

```
x <- c("Rochelle is the Greatest")
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "rochelle is the Greatest"
```

15.1.6 Locales

Above I used `str_to_lower()` to change the text to lower case. You can also use `str_to_upper()` or `str_to_title()`. However, changing case is more complicated than it might at first appear because different languages have different rules for changing case. You can pick which set of rules to use by specifying a locale:

```
# Turkish has two i's: with and without a dot, and it
# has a different rule for capitalising them:
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

The locale is specified as a ISO 639 language code, which is a two or three letter abbreviation. If you don't already know the code for your language, Wikipedia has a good list. If you leave the locale blank, it will use the current locale, as provided by your operating system.

Another important operation that's affected by the locale is sorting. The base R `order()` and `sort()` functions sort strings using the current locale. If you want robust behaviour across different computers, you may want to use `str_sort()` and `str_order()` which take an additional `locale` argument:

```
x <- c("apple", "eggplant", "banana")
str_sort(x, locale = "en")  # English
#> [1] "apple"    "banana"   "eggplant"
str_sort(x, locale = "haw") # Hawaiian
#> [1] "apple"    "eggplant" "banana"
```

Challenges

1. In code that doesn't use `stringr`, you'll often see `paste()` and `paste0()`. What's the difference between the two functions? What `stringr` function

are they equivalent to? How do the functions differ in their handling of `NA`?

2. In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()`.

3. Use `str_length()` and `str_sub()` to extract the middle character from a string. What will you do if the string has an even number of characters?

4. What does `str_trim()` do? What's the opposite of `str_trim()`?

15.2 Regular expressions

Regular expressions are a very terse language that allow you to describe patterns in strings. They take a little while to get your head around, but once you understand them, you'll find them extremely useful.

To learn regular expressions, we'll use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression, and show you how they match. We'll start with very simple regular expressions and then gradually get more and more complicated. Once you've mastered pattern matching, you'll learn how to apply those ideas with various stringr functions.

15.2.1 Basic matches

The simplest patterns match exact strings:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
#> PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

The next step up in complexity is `.`, which matches any character (except a newline):

```
x <- c("apple", "banana", "pear")
str_view(x, ".a.")
```

15.2.2 Escape Characters

If "`.`" matches any character, how do you match the character "`.`"? You need to use an "escape" to tell the regular expression you want to match it exactly, not use its special behaviour.

Regexps use the backslash, `\`, to escape special behaviour. So to match an `.`, you need the regexp `\.`. Unfortunately this creates a problem. We use *strings* to

represent regular expressions, and `\` is also used as an escape symbol in strings. So to create the regular expression `\.` we need the string `"\\."`.

```
# To create the regular expression, we need \\
dot <- "\\."
# But the expression itself only contains one:
writeLines(dot)
#> \.
```

In this lesson, I'll write regular expression as `\.` and strings that represent the regular expression as `"\\."`.

15.2.3 Anchors

By default, regular expressions will match any part of a string. It's often useful to *anchor* the regular expression so that it matches from the start or end of the string. You can use:

- `^` to match the start of the string.
- `$` to match the end of the string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
str_view(x, "a$")
```

To remember which is which, try this mnemonic which I learned from Evan Misshula: if you begin with power (`^`), you end up with money (`$`).

To force a regular expression to only match a complete string, anchor it with both `^` and `$`:

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
str_view(x, "^apple$")
```

15.2.4 Character classes and alternatives

There are a number of special patterns that match more than one character. You've already seen `.`, which matches any character apart from a newline. There are four other useful tools:

- `\d`: matches any digit.
- `\s`: matches any whitespace (e.g. space, tab, newline).
- `[abc]`: matches a, b, or c.
- `[^abc]`: matches anything except a, b, or c.

Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the `\` for the string, so you'll type `"\\d"` or `"\\s"`.

A character class containing a single character is a nice alternative to backslash escapes when you want to include a single metacharacter in a regex. Many people find this more readable.

```
# Look for a literal character that normally has special meaning in a regex
x <- c("abc", "a.c", "a*c", "a c")
str_view(x, "a[.]c")
```

```
str_view(x, ".[*]c")
```

```
str_view(x, "a[ ]")
```

This works for most (but not all) regex metacharacters: `$ . | ? * + ( ) [ {`. Unfortunately, a few characters have special meaning even inside a character class and must be handled with backslash escapes: `] \ ^` and `-`.

You can use *alternation* to pick between one or more alternative patterns. For example, `abc|deaf` will match either '"abc"', or `"deaf"`.

Like with mathematical expressions, if precedence ever gets confusing, use parentheses to make it clear what you want:

```
x <- c("grey", "gray")
str_view(x, "gr(e|a)y")
```

Challenges

Create regular expressions to find all words that:

1. Start with a vowel.
2. That only contain consonants. (Hint: thinking about matching "not"-vowels.)
3. End with `ed`, but not with `eed`.
4. End with `ing` or `ise`.

15.2.5 Repetition

The next step up in power involves controlling how many times a pattern matches:

- `?`: 0 or 1
- `+`: 1 or more
- `*`: 0 or more

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
str_view(x, "CC+")
```

```
str_view(x, 'C[LX]+')
```

15.2.6 Regex Resources

For more information on regular expressions, see:

1. this tutorial
2. this cheatsheet

15.3 Common Tools

Now that you've learned the basics of regular expressions, it's time to learn how to apply them to real problems. In this section you'll learn a wide array of `stringr` functions that let you:

- Determine which strings match a pattern.
- Find the positions of matches.
- Extract the content of matches.
- Replace matches with new values.
- Split a string based on a match.

15.3.1 Detect matches

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input:

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
#> [1]  TRUE FALSE  TRUE
```

Remember that when you use a logical vector in a numeric context, `FALSE` becomes 0 and `TRUE` becomes 1. That makes `sum()` and `mean()` useful if you want to answer questions about matches across a larger vector:

```
# see common words
words <- stringr::words
words[1:10]
#>  [1] "a"        "able"     "about"    "absolute" "accept"   "account"
#>  [7] "achieve"  "across"   "act"      "active"
# How many common words start with t?
```

```
sum(str_detect(words, "^t"))
#> [1] 65
# What proportion of common words end with a vowel?
mean(str_detect(words, "[aeiou]$"))
#> [1] 0.277
```

A common use of `str_detect()` is to select the elements that match a pattern. You can do this with logical subsetting, or the convenient `str_subset()` wrapper:

```
words[str_detect(words, "x$")]
#> [1] "box" "sex" "six" "tax"
str_subset(words, "x$")
#> [1] "box" "sex" "six" "tax"
```

Typically, however, your strings will be one column of a data frame, and you'll want to use filter instead:

```
df <- data.frame(
  i = seq_along(words),
  word = words
)
df %>%
  filter(str_detect(word, "x$"))
#>     i word
#> 1 108  box
#> 2 747  sex
#> 3 772  six
#> 4 841  tax
```

A variation on `str_detect()` is `str_count()`: rather than a simple yes or no, it tells you how many matches there are in a string:

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
#> [1] 1 3 1
# On average, how many vowels per word?
mean(str_count(words, "[aeiou]"))
#> [1] 1.99
```

It's natural to use `str_count()` with `mutate()`:

```
df1 <- df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]")
  )
```

```
head(df1)
#>   i     word vowels consonants
#> 1 1        a      1          0
#> 2 2     able      2          2
#> 3 3    about      3          2
#> 4 4 absolute      4          4
#> 5 5   accept      2          4
#> 6 6  account      3          4
```

Challenges

For each of the following challenges, try solving it by using both a single regular expression, and a combination of multiple `str_detect()` calls.

1. Find all words that start or end with `x`.

2. Find all words that start with a vowel and end with a consonant.

15.3.2 Extract matches

To extract the actual text of a match, use `str_extract()`. To show that off, we're going to need a more complicated example. I'm going to use the Harvard sentences. These are provided in `stringr::sentences`:

```
length(sentences)
#> [1] 720
head(sentences)
#> [1] "The birch canoe slid on the smooth planks."
#> [2] "Glue the sheet to the dark blue background."
#> [3] "It's easy to tell the depth of a well."
#> [4] "These days a chicken leg is a rare dish."
#> [5] "Rice is often served in round bowls."
#> [6] "The juice of lemons makes fine punch."
```

Imagine we want to find all sentences that contain a color. We first create a vector of color names, and then turn it into a single regular expression:

```
colors <- c("red", "orange", "yellow", "green", "blue", "purple")
color_match <- str_c(colors, collapse = "|")
color_match
#> [1] "red|orange|yellow|green|blue|purple"
```

Now we can select the sentences that contain a color, and then extract the color to figure out which one it is:

```
# find sentences with colors
has_color <- str_subset(sentences, color_match)
head(has_color)
#> [1] "Glue the sheet to the dark blue background."
#> [2] "Two blue fish swam in the tank."
#> [3] "The colt reared and threw the tall rider."
#> [4] "The wide road shimmered in the hot sun."
#> [5] "See the cat glaring at the scared mouse."
#> [6] "A wisp of cloud hung in the blue air."

# extract the color
matches <- str_extract(has_color, color_match)
head(matches)
#> [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

Note that `str_extract()` only extracts the first match. This is a common pattern for stringr functions, because working with a single match allows you to use much simpler data structures. To get all matches, use `str_extract_all()`. It returns a list:

```
all_colors <- str_extract_all(has_color, color_match)
all_colors[15:20]
#> [[1]]
#> [1] "red"
#>
#> [[2]]
#> [1] "red"
#>
#> [[3]]
#> [1] "red"
#>
#> [[4]]
#> [1] "blue"
#>
#> [[5]]
#> [1] "red"
#>
#> [[6]]
#> [1] "blue" "red"
```

If you use `simplify = TRUE`, `str_extract_all()` will return a matrix with short matches expanded to the same length as the longest:

```
str_extract_all(has_color, color_match, simplify = TRUE)
#>       [,1]     [,2]
#>  [1,] "blue"   ""
#>  [2,] "blue"   ""
```

```
#>  [3,] "red"    ""
#>  [4,] "red"    ""
#>  [5,] "red"    ""
#>  [6,] "blue"   ""
#>  [7,] "yellow" ""
#>  [8,] "red"    ""
#>  [9,] "red"    ""
#> [10,] "green"  ""
#> [11,] "red"    ""
#> [12,] "red"    ""
#> [13,] "blue"   ""
#> [14,] "red"    ""
#> [15,] "red"    ""
#> [16,] "red"    ""
#> [17,] "red"    ""
#> [18,] "blue"   ""
#> [19,] "red"    ""
#> [20,] "blue"   "red"
#> [21,] "red"    ""
#> [22,] "green"  ""
#> [23,] "red"    ""
#> [24,] "red"    ""
#> [25,] "red"    ""
#> [26,] "red"    ""
#> [27,] "red"    ""
#> [28,] "red"    ""
#> [29,] "green"  ""
#> [30,] "red"    ""
#> [31,] "green"  ""
#> [32,] "red"    ""
#> [33,] "purple" ""
#> [34,] "green"  ""
#> [35,] "red"    ""
#> [36,] "red"    ""
#> [37,] "red"    ""
#> [38,] "red"    ""
#> [39,] "red"    ""
#> [40,] "blue"   ""
#> [41,] "red"    ""
#> [42,] "blue"   ""
#> [43,] "red"    ""
#> [44,] "red"    ""
#> [45,] "red"    ""
#> [46,] "red"    ""
#> [47,] "green"  ""
```

```
#> [48,] "green"  ""
#> [49,] "green"  "red"
#> [50,] "red"    ""
#> [51,] "red"    ""
#> [52,] "yellow" ""
#> [53,] "red"    ""
#> [54,] "orange" "red"
#> [55,] "red"    ""
#> [56,] "red"    ""
#> [57,] "red"    ""
```

Challenges

In the previous example, you might have noticed that the regular expression matched "flickered", which is not a color. Modify the regex to fix the problem.

15.3.3 Replacing matches

`str_replace()` and `str_replace_all()` allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-") # replace the first instance of a match
#> [1] "-pple"  "p-ar"   "b-nana"
str_replace_all(x, "[aeiou]", "-") # replace all instances of a match
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

With `str_replace_all()` you can perform multiple replacements by supplying a named vector:

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
#> [1] "one house"    "two cars"     "three people"
```

15.3.4 Splitting

Use `str_split()` to split a string up into pieces. For example, we could split sentences into words:

```
sentences %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"     "smooth"
```

```
#> [8] "planks."
#>
#> [[2]]
#> [1] "Glue"        "the"         "sheet"       "to"          "the"
#> [6] "dark"        "blue"        "background."
#>
#> [[3]]
#> [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"    "a"     "well."
#>
#> [[4]]
#> [1] "These"   "days"    "a"       "chicken" "leg"     "is"      "a"
#> [8] "rare"    "dish."
#>
#> [[5]]
#> [1] "Rice"   "is"     "often"  "served" "in"     "round"  "bowls."
```

Like the other stringr functions that return a list, you can use `simplify = TRUE` to return a matrix:

```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]    [,2]    [,3]    [,4]      [,5]  [,6]    [,7]      [,8]
#> [1,] "The"   "birch" "canoe" "slid"    "on"  "the"   "smooth" "planks."
#> [2,] "Glue"  "the"   "sheet" "to"      "the" "dark"  "blue"   "background."
#> [3,] "It's"  "easy"  "to"    "tell"    "the" "depth" "of"     "a"
#> [4,] "These" "days"  "a"     "chicken" "leg" "is"    "a"      "rare"
#> [5,] "Rice"  "is"    "often" "served"  "in"  "round" "bowls." ""
#>      [,9]
#> [1,] ""
#> [2,] ""
#> [3,] "well."
#> [4,] "dish."
#> [5,] ""
```

You can also request a maximum number of pieces:

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]      [,2]
#> [1,] "Name"    "Hadley"
#> [2,] "Country" "NZ"
#> [3,] "Age"     "35"
```

Instead of splitting up strings by patterns, you can also split up by character, line, sentence and word `boundary()`s:

```
x <- "This is a sentence.  This is another sentence."
str_view_all(x, boundary("word"))
```

```
str_split(x, boundary("word"))[[1]]
#> [1] "This"     "is"       "a"        "sentence" "This"     "is"       "another"
#> [8] "sentence"
```

Challenges

1. Split up a string like `"apples, pears, and bananas"` into individual components.

2. What does splitting with an empty string (`""`) do? Experiment, and then read the documentation.

15.4 Other types of patterns

When you use a pattern that's a string, it's automatically wrapped into a call to `regex()`:

```
# The regular call:
str_view(fruit, "nana")
# Is shorthand for
str_view(fruit, regex("nana"))
```

You can use the other arguments of `regex()` to control details of the match:

- `ignore_case = TRUE` allows characters to match either their uppercase or lowercase forms. This always uses the current locale.

  ```
  bananas <- c("banana", "Banana", "BANANA")
  str_view(bananas, "banana")
  ```

  ```
  str_view(bananas, regex("banana", ignore_case = TRUE))
  ```

- `multiline = TRUE` allows `^` and `$` to match the start and end of each line rather than the start and end of the complete string.

  ```
  x <- "Line 1\nLine 2\nLine 3"
  str_extract_all(x, "^Line")[[1]]
  #> [1] "Line"
  str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
  #> [1] "Line" "Line" "Line"
  ```

15.4.1 stringi

stringr is built on top of the **stringi** package. stringr is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions. stringi, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need.

If you find yourself struggling to do something in stringr, it's worth taking a look at stringi. The packages work very similarly, so you should be able to translate your stringr knowledge in a natural way. The main difference is the prefix: `str_` vs. `stri_`.

Challenges

Find the stringi functions that:

1. Count the number of words.
2. Find duplicated strings.
3. Generate random text.

Acknowledgments

This page was adapted from the following sources:

1. R for Data Science licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0

Chapter 16

Programming in R

This unit covers some more advanced programming in R - namely:

1. Conditional Flow
2. Functions
3. Iteration

Mastering these skills will make you virtually invincible in R!

Note that these concepts are **not specific to R**. While the syntax might vary, the basic idea of flow, functions, and iteration are common across all scripting languages. So if you ever think of picking up Python or something else, it's critical to familiarize yourself with these concepts.

16.1 Conditional Flow

Sometimes you only want to execute code if a certain condition is met. To do that, we use an **if-else statement.** It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

`condition` is a statement that must always evaluate to either `TRUE` or `FALSE`. This is similar to `filter()`, except `condition` can only be a single value (i.e. a vector of length 1), whereas `filter()` works for entire vectors (or columns).

Let's look at a simple example:

```
age = 84
if (age > 60) {
    print("OK Boomer")
} else {
    print("But you don't look like a professor!")
}
#> [1] "OK Boomer"
```

We refer to the first `print` command as the first *branch*.

Let's change the `age` variable to execute the second branch:

```
age = 20
if (age > 60) {
    print("OK Boomer")
} else {
    print("But you don't look like a professor!")
}
#> [1] "But you don't look like a professor!"
```

16.1.1 Multiple Conditions

You can chain conditional statements together:

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  # do something completely different
}
```

16.1.2 Complex Statements

We can generate more complex conditional statements with boolean operators like `&` and `|`:

```
age = 45

if (age > 60) {
    print("OK Boomer")
} else if (age < 60 & age > 40) {
    print("How's the midlife crisis?")
} else {
    print("But you don't look like a professor!")
```

```
}
#> [1] "How's the midlife crisis?"
```

16.1.3 Code Style

Both `if` and `function` should (almost) always be followed by squiggly brackets (`{}`), and the contents should be indented. This makes it easier to see the hierarchy in your code by skimming the left-hand margin.

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it's followed by else. Always indent the code inside curly braces.

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

if (y == 0) {
  log(x)
}
else {
  y ^ x
}

# Good
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

16.1.4 `if` vs. `if_else`

Because if-else conditional statements like the ones outlined above must always resolve to a single `TRUE` or `FALSE`, they cannot be used for vector operations. Vector operations are where you make multiple comparisons simultaneously for each value stored inside a vector.

Consider the `gapminder` data and imagine you wanted to create a new column identifying whether or not a country-year observation has a life expectancy of at least 35.

```
gap <- read.csv("Data/gapminder-FiveYearData.csv")
head(gap)
#>       country year      pop continent lifeExp gdpPercap
#> 1 Afghanistan 1952  8425333      Asia    28.8       779
#> 2 Afghanistan 1957  9240934      Asia    30.3       821
#> 3 Afghanistan 1962 10267083      Asia    32.0       853
#> 4 Afghanistan 1967 11537966      Asia    34.0       836
#> 5 Afghanistan 1972 13079460      Asia    36.1       740
#> 6 Afghanistan 1977 14880372      Asia    38.4       786
```

This sounds like a classic if-else operation. For each observation, if `lifeExp` is greater than or equal to `35`, then the value in the new column should be `1`. Otherwise, it should be `0`. But what happens if we try to implement this using an if-else operation like above?

```
gap_if <- gap %>%
   mutate(life.35 = if(lifeExp >= 35){
     1
   } else {
     0
   })
#> Warning in if (lifeExp >= 35) {: the condition has length > 1 and only the first
#> element will be used

head(gap_if)
#>       country year      pop continent lifeExp gdpPercap life.35
#> 1 Afghanistan 1952  8425333      Asia    28.8       779       0
#> 2 Afghanistan 1957  9240934      Asia    30.3       821       0
#> 3 Afghanistan 1962 10267083      Asia    32.0       853       0
#> 4 Afghanistan 1967 11537966      Asia    34.0       836       0
#> 5 Afghanistan 1972 13079460      Asia    36.1       740       0
#> 6 Afghanistan 1977 14880372      Asia    38.4       786       0
```

This did not work correctly. Because `if()` can only handle a single `TRUE/FALSE` value, it only checked the first row of the data frame. That row contained `28.801`, so it generated a vector of length 1704 with each value being `0`.

Because we in fact want to make this if-else comparison 1704 times, we should instead use **`if_else()`**. This **vectorizes** the if-else comparison and makes a separate comparison for each row of the data frame. This allows us to correctly generate this new column.

```
gap_ifelse <- gap %>%
  mutate(life.35 = if_else(lifeExp >= 35, 1, 0))

head(gap_ifelse)
#>       country year      pop continent lifeExp gdpPercap life.35
```

```
#> 1 Afghanistan 1952  8425333      Asia    28.8       779       0
#> 2 Afghanistan 1957  9240934      Asia    30.3       821       0
#> 3 Afghanistan 1962 10267083      Asia    32.0       853       0
#> 4 Afghanistan 1967 11537966      Asia    34.0       836       0
#> 5 Afghanistan 1972 13079460      Asia    36.1       740       1
#> 6 Afghanistan 1977 14880372      Asia    38.4       786       1
```

16.2 Functions

Functions are the basic building blocks of programs. Think of them as "mini-scripts" or "tiny commands." We've already used dozens of functions created by others (e.g. `filter()`, `mean()`.)

This lesson teaches you how to write you own functions, and why you would want to do so. The details are pretty simple, but this is one of those ideas where it's good to get lots of practice!

16.2.1 Why Write Functions?

Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. For example, take a look at the following code:

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))
df$b <- (df$b - min(df$b)) / (max(df$b) - min(df$a))
df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))
df$d <- (df$d - min(df$d)) / (max(df$d) - min(df$d))
```

You might be able to puzzle out that this rescales each column to have a range from 0 to 1. But did you spot the mistake? I made an error when copying-and-pasting the code for `df$b`: I forgot to change an `a` to a `b`.

Functions have a number of advantages over this "copy-and-paste" approach:

- **They are easy to reuse.** If you need to change things, you only have to update code in one place, instead of many.
- **They are self-documenting.** Functions name pieces of code the way variables name strings and numbers. Give your function a good name and

you will easily remember the function and its purpose.

- **They are easier to debug.** There are fewer chances to make mistakes because the code only exists in one location (i.e. updating a variable name in one place, but not in another).

16.2.2 Anatomy of a Function

Functions have three key components:

1. A **name**. This should be informative and describe what the function does.
2. The **arguments**, or list of inputs, to the function. They go inside the parentheses in `function()`.
3. The **body**. This is the block of code within `{}` that immediately follows `function(...)`, and is the code that you develop to perform the action described in the name using the arguments you provide.

```
my_function <- function(x, y){
  # do
  # something
  # here
  return(result)
}
```

In this example, `my_function` is the **name** of the function, `x` and `y` are the **arguments**, and the stuff inside the `{}` is the **body**.

16.2.3 Writing a Function

Let's re-write the scaling code above as a function. To write a function you need to first analyze the code. How many inputs does it have?

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

(df$a - min(df$a)) / (max(df$a) - min(df$a))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612 0.601
```

This code only has one input: `df$a`. To make the inputs more clear, it's a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, which I'll call `x`:

```
x <- df$a
(x - min(x)) / (max(x) - min(x))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612 0.601
```

There is some duplication in this code. We're computing the range of the data three times, so it makes sense to do it in one step:

```
rng <- range(x)
rng
#> [1] -2.44  1.15

(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612 0.601
```

Pulling out intermediate calculations into named variables is a good practice because it becomes more clear what the code is doing. Now that I've simplified the code, and checked that it still works, I can turn it into a function:

```
rescale01 <- function(x) {
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  return(scaled)
}
```

Note the overall process: I only made the function after I'd figured out how to make it work with a simple input. It's easier to start with working code and turn it into a function; it's harder to create a function and then try to make it work.

At this point it's a good idea to check your function with a few different inputs:

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0

rescale01(c(1, 2, 3, 5))
#> [1] 0.00 0.25 0.50 1.00
```

16.2.4 Using a Function

Two important points about using (or *calling**) functions:

1. Notice that when we **call** a function, we're passing a value into it that is assigned to the parameter we defined when writing the function. In this case, the parameter `x` is automatically assigned to `c(-10, 0, 10)`.
2. When using functions, by default the returned object is merely printed to the screen. If you want it saved, you need to assign it to an object.

Let's see if we can simplify the original example with our brand new function:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Compared to the original, this code is easier to understand and we've eliminated one class of copy-and-paste errors. There is still quite a bit of duplication since we're doing the same thing to multiple columns. We'll learn how to eliminate that duplication in the lesson on iteration.

Another advantage of functions is that if our requirements change, we only need to make the change in one place. For example, we might discover that some of our variables include `NA` values, and `rescale01()` fails:

```
rescale01(c(1, 2, NA, 3, 4, 5))
#> [1] NA NA NA NA NA NA
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(1, 2, NA, 3, 4, 5))
#> [1] 0.00 0.25   NA 0.50 0.75 1.00
```

16.2.5 Variable Scope

Analyze the following function:

1. Identify the name, arguments, and body
2. What does it do?
3. If `a = 3` and `b = 4`, what should we expect the output to be?

```
pythagorean <- function(a, b){
  hypotenuse <- sqrt(a^2 + b^2)
  return(hypotenuse)
}
```

Now take a look at the following code:

```
pythagorean(a = 3, b = 4)
#> [1] 5

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

Why does this generate an error? Why can we not see the results of `hypotenuse`? After all, it was generated by `pythagorean`, right?

When you call a function, a temporary workspace is set up that will be destroyed when the function returns, either by:

1. getting to the end, or
2. an explicit return statement

So think of functions as an alternative reality, where objects are created and destroyed in a function call.

This is why you do not see hypotenuse listed in the environment - it has already been destroyed.

Global vs. Local Environments

Things can get confusing when you use the same names for variables both inside and outside a function. Check out this example:

```
pressure = 103.9
adjust <- function(t){
    temperature = t * 1.43 / pressure
    return(temperature)
}
pressure
#> [1] 104
temperature
#> Error in eval(expr, envir, enclos): object 'temperature' not found
```

`t` and `temperature` are **local** variables in `adjust`. Local variables are:

- Defined in the function.
- Not visible in the main program.
- Remember: a function parameter is a variable that is automatically assigned a value when the function is called.

`pressure` is a **global** variable. Global variables are:

- Defined outside any particular function.
- Visible everywhere.

This difference is referred to as **scope**. The **scope** of a variable is the part of a program that can 'see' that variable.

16.2.6 Arguments

Functions do not need to take input.

```
print_hello <- function(){
    print("hello")
}
print_hello()
#> [1] "hello"
```

But if a function takes input, arguments can be passed to functions in different ways.

1) **Positional arguments** are mandatory and have no default values.

```
send <- function(message, recipient){
  message <- paste(message, recipient)
  return(message)
}
send("Hello", "world")
#> [1] "Hello world"
```

In the case above, it is possible to use argument **names** when calling the functions and, by doing so, it is possible to switch the order of arguments. For instance:

```
send(recipient='World', message='Hello')
#> [1] "Hello World"
```

However, positional arguments (`send('Hello', 'World')`) are greatly perfered over names (`send(recipient='World', message='Hello')`), as it is very easy to accidentally specifying incorrect argument values.

2) **Keyword arguments** are not mandatory and have default values. They are often used for optional parameters sent to the function.

```
send <- function(message, recipient, cc=NULL){
  message <- paste(message, recipient, "cc:", cc)
  return(message)
}
send("Hello", "world")
#> [1] "Hello world cc: "
send("Hello", "world", "rochelle")
#> [1] "Hello world cc: rochelle"
```

Here `cc` and bcc are `optional`, and evaluate to `NULL` when they are not passed another value.

16.2.7 Challenges

Challenge 1

Write a function that calculates the sum of the squared value of two numbers. For instance, it should generate the following output:

```
my_function(3, 4)
# [1] 25
```

Challenge 2

Write `both_na()`, a function that takes two vectors of the same length and returns the number of positions that have an NA in both vectors.

Challenge 3

Fill in the blanks to create a function that takes a name like "Rochelle Terman" and returns that name in uppercase and reversed, like "TERMAN, ROCHELLE"

```
standard_names <- function(name){
    upper_case = toupper(____) # make upper
    upper_case_vec = strsplit(_____, split = ' ')[[1]] # turn into a vector
    first_name = ______ # take first name
    last_name = _______ # take last name
    reversed_name = paste(______, _______, sep = ", ") # reverse and separate by a comma and spac
    return(reversed_name)
}
```

Challenge 4

Look at the following function:

```
print_date <- function(year, month, day){
    joined = paste(as.character(year), as.character(month), as.character(day), sep = "/")
    return(joined)
}
```

What does this short program print?

```
print_date(day=1, month=2, year=2003)
```

Acknowledgements and Resources

- R for Data Science.
- Computing for Social Sciences

16.3 Iteration

In the last unit, we talked about how important it is to reduce duplication in your code by creating functions instead of copying-and-pasting. Avoiding duplication allows for more readable, more flexible, and less error-prone code.

Functions are one method of reducing duplication in your code. Another tool for reducing duplication is **iteration**, which lets you do the same task to multiple inputs.

In this chapter you'll learn about four approaches to iteratation:

1. Vectorized functions
2. For-loops
3. `map` and functional programming
4. Scoped verbs in `dplyr`

16.3.1 Vectorized Functions

Most of R's built-in functions are **vectorized**, meaning that the function will operate on all elements of a vector without needing to loop through and act on each element at a time.

That means you should never need to perform explicit iteration when performing simple mathematical computations.

```
x <- 1:4
x * 2
#> [1] 2 4 6 8
```

Notice that the multiplication happened to each element of the vector. Most built-in functions also operate element-wise on vectors:

```
x <- 1:4
log(x)
#> [1] 0.000 0.693 1.099 1.386
```

We can also add two vectors together:

```
x <- 1:4
y <- 6:9
```

```
x + y
#> [1]  7  9 11 13
```

Notice that each element of x was added to its corresponding element of y:

```
x:  1  2  3  4
    +  +  +  +
y:  6  7  8  9
---------------
    7  9 11 13
```

What happens if you add two vectors of different lengths?

```
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
```

Here, R will expand the shortest vector to the same length as the longest. This is called **recycling**. This usually (but not always) happens silently, meaning R will not warn you. Beware!

16.3.2 For-loops

You will frequently need to iterate over vectors or data frames, perform an operation on each element, and save the results somewhere.

For example, imagine we have this simple dataframe:

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

We want to compute the median of each column. You *could* do with copy-and-paste:

```
median(df$a)
#> [1] -0.246
median(df$b)
#> [1] -0.287
median(df$c)
#> [1] -0.0567
median(df$d)
#> [1] 0.144
```

But that breaks our rule of thumb: never copy and paste more than twice. Instead, we could use a `for` loop:

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[i] <- median(df[[i]])        # 3. body
}
output
#> [1] -0.2458 -0.2873 -0.0567  0.1443
```

Components of a `for` Loop

Every `for` loop has three components:

1. The **output**: `output <- vector("double", length(x))`.

Before you start a loop, you need to create an empty vector to store the output of the loop. Notice that the object is created **outside** the loop!

Preallocating space for your output is very important for efficiency: if you grow the `for` loop at each iteration using `c()` (for example), your `for` loop will be very slow.

2. The **sequence**: `i in seq_along(df)`.

This determines what to loop over. In this case, the sequence is `seq_along(df)`, which creates a numeric vector for a sequence of numbers beginning at 1 and continuing until it reaches the length of `df` (the length here is the number of columns in `df`).

```
seq_along(df)
#> [1] 1 2 3 4
```

It's useful to think of `i` as a pronoun, like "it". Each iteration of the `for` loop will assign `i` to a new value based on the designed sequence:

```
| Iteration | i =  |
|-----------|------|
| 1         | 1    |
| 2         | 2    |
| 3         | 3    |
| 4         | 4    |
=
```

NB: `seq_along` is a safe version of the more familiar `1:length(l)`, with an important difference: if you have a zero-length vector, `seq_along()` does the right thing:

```
y <- vector("double", 0)
seq_along(y)
#> integer(0)
```

```
1:length(y)
#> [1] 1 0
```

You probably won't create a zero-length vector deliberately, but it's easy to create one accidentally. If you use `1:length(x)` instead of `seq_along(x)`, you're likely to get a confusing error message.

3. The **body**: `output[[i]] <- median(df[[i]])`.

This is the code that does the work. It runs repeatedly, each time with a different value for `i`:

```
| Iteration | i =  | body                          |
|-----------|------|-------------------------------|
| 1         | 1    | output[[1]] <- median(df[[1]]) |
| 2         | 2    | output[[2]] <- median(df[[2]]) |
| 3         | 3    | output[[3]] <- median(df[[3]]) |
| 4         | 4    | output[[4]] <- median(df[[4]]) |
```

NB:: We use `[[` notation to reference each column of `df` using indices of columns, instead of `$` and column names.

16.3.3 Challenges

Challenge 1.

Fill in the blanks to write a `for` loop that calculates the arithmetic mean for every column in `mtcars`.

```
mtcars.means <- vector("double", ______)
for(i in ______){
  ______[i] <- mean(______[[i]])
}
```

Challenge 2.

Check out the `iris` dataset:

```
kable(head(iris))
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Write a `for` loop that calculates the number of unique values in each column of `iris`. Before you write the `for` loop, identify the three components you need:

1. Output
2. Sequence
3. Body

Challenge 3.

Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100. Store them in a list.

16.3.4 Functional Programming and `map`

Loops are not as important in R as they are in other languages because R is a **functional** programming language. This means that it's possible to wrap up `for` loops in a function, and call that function instead of using the for loop directly.

The pattern of looping over a vector, doing something to each element and saving the results is so common that the `purrr` package provides a family of functions to do it for you. They effectively eliminate the need for many common `for` loops.

There is one function for each type of output:

1. `map()` makes a list.
2. `map_lgl()` makes a logical vector.
3. `map_int()` makes an integer vector.
4. `map_dbl()` makes a double vector.
5. `map_chr()` makes a character vector.

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that's the same length (and has the same names) as the input.

NB: Some people will tell you to avoid `for` loops because they are slow. They're wrong! (Well, at least they're rather out of date, as for loops haven't been slow for many years). The main benefit of using functions like `map()` is not speed, but clarity: they make your code easier to write and to read.

To see how `map` works, consider (again) this simple data frame:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
```

```
  d = rnorm(10)
)
```

What if we wanted to calculate the mean, median, and standard deviation of each column?

```
map_dbl(df, mean)
#>      a      b      c      d
#>  0.116  0.127 -0.089  0.281
map_dbl(df, median)
#>       a       b       c       d
#>  0.0583  0.0244 -0.0571  0.2604
map_dbl(df, sd)
#>     a     b     c     d
#> 1.161 1.226 1.024 0.798
```

Compared to using a for loop, this approach is much easier to read, and less error-prone.

The data can even be piped!

```
df %>% map_dbl(mean)
#>      a      b      c      d
#>  0.116  0.127 -0.089  0.281
df %>% map_dbl(median)
#>       a       b       c       d
#>  0.0583  0.0244 -0.0571  0.2604
df %>% map_dbl(sd)
#>     a     b     c     d
#> 1.161 1.226 1.024 0.798
```

We can also pass additional arguments. For example, the function `mean` passes an optional argument `trim`. From the help file: "the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the `mean`is computed.

```
map_dbl(df, mean, trim = 0.5)
#>       a       b       c       d
#>  0.0583  0.0244 -0.0571  0.2604
```

Check out other fun applications of `map` functions here

16.3.5 Challenges

Write code that uses one of the map functions to:

1. Calculates the arithmetic mean for every column in `mtcars`.
2. Calculates the number of unique values in each column of `iris`.

3. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

16.3.6 Scoped Verbs

The last iteration technique we'll discuss is scoped verbs in `dplyr`.

Frequently, when working with dataframes, we want to apply a function to multiple columns. For example, let's say we want to calculate the mean value of each column in `mtcars`.

If we wanted to calculate the average of a single column, it would be pretty simple using just regular `dplyr` functions:

```
mtcars %>%
  summarize(mpg = mean(mpg))
#>    mpg
#> 1 20.1
```

But if we want to calculate the mean for all of them, we'd have to duplicate this code many times over:

```
mtcars %>%
  summarize(mpg = mean(mpg),
            cyl = mean(cyl),
            disp = mean(disp),
            hp = mean(hp),
            drat = mean(drat),
            wt = mean(wt),
            qsec = mean(qsec),
            vs = mean(vs),
            am = mean(am),
            gear = mean(gear),
            carb = mean(carb))
#>    mpg  cyl disp  hp drat   wt qsec    vs    am gear carb
#> 1 20.1 6.19  231 147  3.6 3.22 17.8 0.438 0.406 3.69 2.81
```

This is very repetitive and prone to mistakes!

We just saw one approach to solve this problem: `map`. Another approach is **scoped verbs**.

Scoped verbs allow you to use standard verbs (or functions) in `dplyr` that affect multiple variables at once.

- `_if` allows you to pick variables based on a predicate function like `is.numeric()` or `is.character()`
- `_at` allows you to pick variables using the same syntax as `select()`
- `_all` operates on all variables

These verbs can apply to `summarize`, `filter`, or `mutate`. Let's go over `summarize`:

`summarize_all()`

`summarize_all()` takes a dataframe and a function and applies that function to each column:

```
mtcars %>%
  summarize_all(.funs = mean)
#>    mpg  cyl disp  hp drat   wt qsec    vs    am gear carb
#> 1 20.1 6.19  231 147  3.6 3.22 17.8 0.438 0.406 3.69 2.81
```

`summarize_at()`

`summarize_at()` allows you to pick columns in the same way as `select()`, that is, based on their names. There is one small difference: you need to wrap the complete selection with the `vars()` helper (this avoids ambiguity).

```
mtcars %>%
  summarize_at(.vars = vars(mpg, wt), .funs = mean)
#>    mpg   wt
#> 1 20.1 3.22
```

`summarize_if()`

`summarize_if()` allows you to pick variables to summarize based on some property of the column. For example, what if we want to apply a numeric summary function only to numeric columns:

```
iris %>%
  summarize_if(.predicate = is.numeric, .funs = mean)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1         5.84        3.06         3.76         1.2
```

`mutate` and `filter` work in a similar way. To see more, check out Scoped verbs by the Data Challenge Lab

Acknowledgments

A good portion of this lesson is based on:

- R for Data Science
- Computing for Social Sciences

Chapter 17

Collecting Data from the Web

17.1 Introduction

There's a ton of web data that's useful to social scientists, including:

- social media
- news media
- government publications
- organizational records

There are two ways to get data off the web:

1. **Web APIs** - i.e. application-facing, for computers
2. **Webscraping** - i.e. user-facing websites for humans

Rule of Thumb: Check for API first. If not available, scrape.

17.2 Web APIs

API stands for **Application Programming Interface**. Broadly defined, an API is a set of rules and procedures that facilitate interactions between computers and their applications.

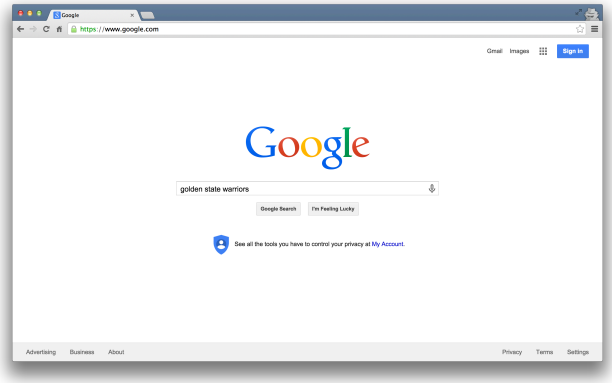
A very common type of API is the **Web API**, which (among other things) allows users to query a remote database over the internet.

Web APIs take on a variety of formats, but the vast majority adhere to a particular style known as **Reperesentational State Transfer** or **REST**. What

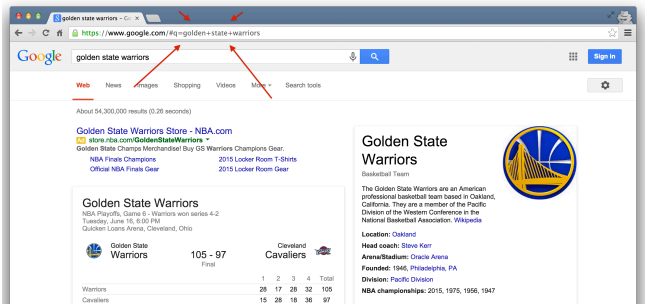
makes these "RESTful" APIs so convenient is that we can use them to query databases using URLs.

RESTful Web APIs are All Around You...

Consider a simple Google search:



Ever wonder what all that extra stuff in the address bar was all about? In this case, the full address is Google's way of sending a query to its databases requesting information related to the search term "golden state warriors".



In fact, it looks like Google makes its query by taking the search terms, separating each of them with a "+", and appending them to the link "https://www.google.com/#q=". Therefore, we should be able to actually change our Google search by adding some terms to the URL and following the general format...

Learning how to use RESTful APIs is all about learning how to format these URLs so that you can get the response you want.

17.2.1 Some Basic Terminology

Let's get on the same page with some basic terminology:

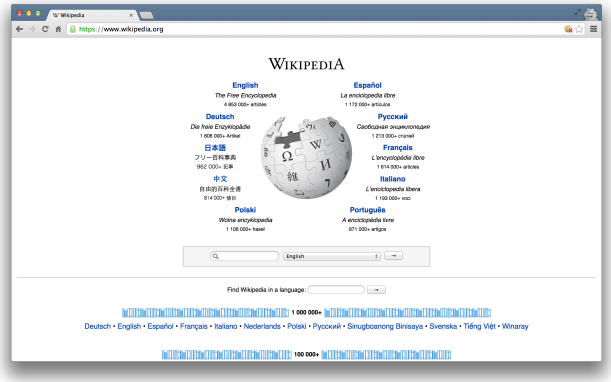
- **Uniform Resource Location (URL)**: a string of characters that, when interpreted via the Hypertext Transfer Protocol (HTTP), points to a data resource, notably files written in Hypertext Markup Language (HTML) or a subset of a database. This is often referred to as a "call".
- **HTTP Methods/Verbs**:
  - *GET*: requests a representation of a data resource corresponding to a particular URL. The process of executing the GET method is often referred to as a "GET request" and is the main method used for querying RESTful databases.
  - *HEAD*, *POST*, *PUT*, *DELETE*: other common methods, though mostly never used for database querying.

17.2.2 How Do GET Requests Work?

A Web Browsing Example

As you might suspect from the example above, surfing the web is basically equivalent to sending a bunch of `GET` requests to different servers and asking for different files written in HTML.

Suppose, for instance, I wanted to look something up on Wikipedia. My first step would be to open my web browser and type in http://www.wikipedia.org. Once I hit return, I'd see the page below.



Several different processes occured, however, between me hitting "return" and the page finally being rendered. In order:

1. The web browser took the entered character string and used the command-line tool "Curl" to write a properly formatted HTTP GET request and submitted it to the server that hosts the Wikipedia homepage.

2. After receiving this request, the server sent an HTTP response, from which Curl extracted the HTML code for the page (partially shown below).

3. The raw HTML code was parsed and then executed by the web browser, rendering the page as seen in the window.

```
#> No encoding supplied: defaulting to UTF-8.
#> [1] "<!DOCTYPE html>\n<html lang=\"mul\" class=\"no-js\">\n<head>\n<meta charset=\"u
```

Web Browsing as a Template for RESTful Database Querying

The process of web browsing described above is a close analogue for the process of database querying via RESTful APIs, with only a few adjustments:

1. While the Curl tool will still be used to send HTML GET requests to the servers hosting our databases of interest, the character string that we supply to Curl must be constructed so that the resulting request can be interpreted and succesfully acted upon by the server. In particular, it is likely that the character string must encode **search terms and/or filtering parameters**, as well as one or more **authentication codes**. While the terms are often similar across APIs, most are API-specific.

2. Unlike with web browsing, the content of the server's response that is extracted by Curl is unlikely to be HTML code. Rather, it will likely be **raw text response that can be parsed into one of a few file formats commonly used for data storage**. The usual suspects include .csv, .xml, and .json files.

3. Whereas the web browser capably parsed and executed the HTML code, **one or more facilities in R, Python, or other programming languages will be necessary for parsing the server response and converting it into a format for local storage** (e.g. matrices, dataframes, databases, lists, etc.).

17.2.3 Finding APIs

More and more APIs pop up every day. Programmable Web offers a running list of APIs. This list provides a list of APIs that may be useful to Political Scientists.

Here are some APIs that may be useful to you:

- NYT Article API: Provides metdata (title, summaries, dates, etc) from all New York Times articles in their archive.
- GeoNames geographical database: Provides lots of geographical information for all countries and other locations. The `geonames` package provides a wrapper for R.
- The Manifesto Project: Provides text and other information on political party manifestos from around the world. It currently covers over 1000 parties from 1945 until today in over 50 countries on five continents. The `manifestoR` package provides a wrapper for R.
- The Census Bureau: Provides datasets from US Census Bureau. The `tidycensus` package allows users to interface with the US Census Bureau's decennial Census and five-year American Community APIs.

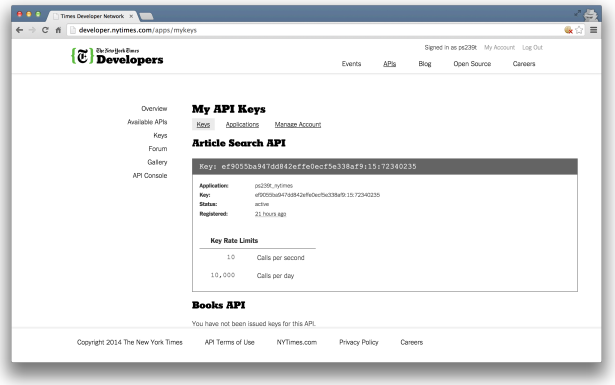
17.2.4 Getting API Access

Most APIs requires a key or other user credentials before you can query their database.

Getting credentialized with a API requires that you register with the organization. Most APIs are set up for developers, so you'll likely be asked to register an "application". All this really entails is coming up with a name for your app/bot/project, and providing your real name, organization, and email. Note that some more popular APIs (e.g. Twitter, Facebook) will require additional information, such as a web address or mobile number.

Once you've successfully registered, you will be assigned one or more keys, tokens, or other credentials that must be supplied to the server as part of any API call you make. To make sure that users aren't abusing their data access privileges (e.g. by making many rapid queries), each set of keys will be given **rate limits** governing the total number of calls that can be made over certain intervals of time.

For example, the NYT Article API has relatively generous rate limits — 4,000 requests per day and 10 requests per minute. So we need to "sleep"" 6 seconds between calls to avoid hitting the per minute rate limit.



17.2.5 Using APIs in R

There are two ways to collect data through APIs in R.

1. **Plug-n-play packages**

Many common APIs are available through user-written R Packages. These packages offer functions that "wrap" API queries and format the response. These packages are usually much more convenient than writing our own query, so it's worth searching around for a package that works with the API we need.

2. **Writing our own API request**

If no wrapper function is available, we have to write our own API request, and format the response ourselves using R. This is trickier, but definitely do-able.

17.3 Collecting Twitter Data with RTweet

Twitter actually has two separate APIs:

1. The **REST API** allows you to read and write Twitter data. For research purposes, this allows you to search the recent history of tweets and look up specific users.
2. The **Streaming API** allows you to access public data flowing through Twitter in real-time. It requires your R session to be running continuously, but allows you to capture a much larger sample of tweets while avoiding rate limits for the REST API.

There are several packages for R for accessing and searching Twitter. In this unit, we'll practice using the `RTweet` library, which allows us to easily collect data from Twitter's REST and stream APIs.

17.3.1 Setting up `RTweet`

To use `RTweet`, follow these steps:

1. If you don't have a Twitter account, create one here.
2. Install the `RTweet` package from CRAN.
3. Load the package into R.
4. Send a request to Twitter's API by calling any of the package's functions, like `search_tweets` or`get_timeline`.
5. Approve the browser popup (to authorize the `rstats2twitter` app).
6. Now, you're ready to use RTweet!

Let's go ahead and load `RTweet` along with some other helpful functions:

```
library(tidyverse)
library(rtweet)
library(lubridate)
library(kableExtra)
```

17.3.2 UChicago Political Science Prof Tweets

Let's explore the `RTweet` package to see what we can learn about the tweeting habits of UChicago Political Science faculty.

The function `get_timeline` will pull the most recent `n` number of tweets from a given handle(s). To pull tweets from multiple handles, write out a vector of the handles in the `user` argument.

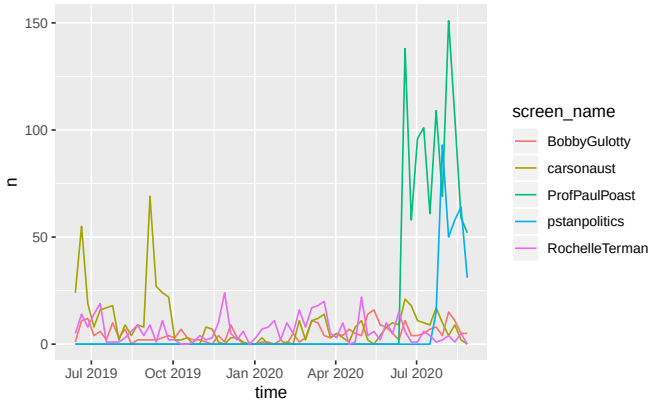
Let's pull tweets from five faculty members in the department.

```
profs <- get_timeline(
  user = c("carsonaust", "profpaulpoast", "pstanpolitics", "rochelleterman", "bobbygulotty"),
  n = 1000
)
kable(head(profs))
```

user_id	status_id	created_at	screen_name	text
805833715	1298318922006114305	2020-08-25 17:58:47	carsonaust	Job alert! Wonderful news, Cornell is
805833715	1296419745508536320	2020-08-20 12:12:08	carsonaust	We are hiring at Wellesley College for
805833715	1295698950117367815	2020-08-18 12:27:57	carsonaust	Useful reminder of the value in investig
805833715	1295461570890420224	2020-08-17 20:44:41	carsonaust	Just recorded all my lectures for Quan
805833715	1294662420707893249	2020-08-15 15:49:09	carsonaust	Intel nugget: passport numbers helped
805833715	1294660982871068673	2020-08-15 15:43:26	carsonaust	US privately warned Russia about bou

Now, let's visualize which professors are tweeting the most, by week.

```
profs %>%
  group_by(screen_name) %>%
  mutate(created_at = as.Date(created_at)) %>%
  filter(created_at >= "2019-06-15") %>%
  ts_plot(by = "week")
```



17.3.3 Hashtags and Text Strings

We can also use `RTweet` to explore certain hashtags or text strings.

Let's take Duke Ellington again – we can use `search_tweets` to pull the most recent n number of tweets that include the hashtag `#DukeEllington` or the string `"Duke Ellington"`.

Hashtag Challenge

Using the documentation for `search_tweets` as a guide, try pulling the 2,000 most recent tweets that include `#DukeEllington` or `"Duke Ellington"` – be sure to exclude retweets from the query.

1. Why didn't your query return 2,000 results?
2. Identify the user that has used either the hashtag or the string in the greatest number of tweets – where is this user from?

```
duke <- search_tweets(
  q = '#DukeEllington OR "Duke Ellington"',
  n = 2000,
  include_rts = FALSE
)
```

```
duke %>%
  group_by(user_id, location) %>%
  summarise(n = n()) %>%
  arrange(desc(n))
#> # A tibble: 565 x 3
#> # Groups:   user_id [565]
#>   user_id             location               n
#>   <chr>               <chr>              <int>
#> 1 1260813006          Redmond, WA           75
#> 2 764290456599396353  Rochester, New York   38
#> 3 1048404121206714369 ""                    22
#> 4 1253135288009777152 New York City         14
#> 5 578607700           New York City         14
#> 6 86155540            New York, New York    14
#> # ... with 559 more rows
```

17.4 Writing API Queries

If no wrapper package is available, we have to write our own API query, and format the response ourslves using R. This is trickier, but definitely do-able.

In this unit, we'll practice constructing our own API queries using the New York Time's `Article API`. This API provides metadata (title, date, summary, etc) on all of New York Times articles.

Fortunately, this API is very well documented!

You can even try it out here.

Load the following packages to get started:

```
library(tidyverse)
library(httr)
library(jsonlite)
library(lubridate)
```

17.4.1 Constructing the API GET Request

Likely the most challenging part of using web APIs is learning how to format your GET request URLs. While there are common architectures for such URLs, each API has its own unique quirks. For this reason, carefully reviewing the API documentation is critical.

Most GET request URLs for API querying have three or four components:

1. *Authenication Key/Token*: a user-specific character string appended to a base URL telling the server who is making the query; allows servers to efficiently manage database access

2. *Base URL*: a link stub that will be at the beginning of all calls to a given API; points the server to the location of an entire database

3. *Search Parameters*: a character string appended to a base URL that tells the server what to extract from the database; basically a series of filters used to point to specific parts of a database

4. *Response Format*: a character string indicating how the response should be formatted; usually one of .csv, .json, or .xml

Let's go ahead and store the these values as variables:

```
key <- "OnzOBobMTn2IRJ7krcT5RXHknkGLqiaI"
base.url <- "http://api.nytimes.com/svc/search/v2/articlesearch.json"
search_term <- "John Mearsheimer"
```

How did I know the `base.url`? I read the documentation.. Notice that this `base.url` also includes the *response format*(`.jston`), so we don't need to configure that directly.

We're ready to make the request. We can use the `GET` function from the `httr` package (another `tidyverse` package) to make an HTTP GET Request.

```
r <- GET(base.url, query = list(`q` = search_term,
                                `api-key` = key))
```

Now, we have an object called `r`. We can get all the information we need from this object. For instance, we can see that the URL has been correctly encoded by printing the URL. Click on the link to see what happens.

```
r$url
#> [1] "http://api.nytimes.com/svc/search/v2/articlesearch.json?q=John%20Mearsheimer&a
```

Challenge 1: Adding a date range

What if we only want to search within a particular date range? The NYT Article Api allows us to specify start and end dates.

Alter the `get.request` code above so that the request only searches for articles in the year 2005.

You're gonna need to look at the documentation here to see how to do this.

Challenge 2: Specifying a results page

The above will return the first 10 results. To get the next ten, you need to add a "page" parameter. Change the search parameters above to get the second 10 resuls.

17.4.2 Parsing the response

We can read the content of the server's response using the `content()` function.

```
response <- httr::content(r, "text")
substr(response, start = 1, stop = 1000)
#> [1] "{\"status\":\"OK\",\"copyright\":\"Copyright (c) 2020 The New York Times Company. All Rig
```

What you see here is JSON text, encoded as plain text. JSON stands for "Javascript object notation." Think of JSON like a nested array built on key/value pairs.

We want to convert the results from JSON format to something easier to work with – notably a data frame.

The **jsonlite** package provides several easy conversion functions for moving between JSON and vectors, data.frames, and lists. Let's use the function `fromJSON` to convert this response into something we can work with:

```
# Convert JSON response to a dataframe
response_df <- fromJSON(response, simplifyDataFrame = TRUE, flatten = TRUE)

# Inspect the dataframe
str(response_df, max.level = 2)
#> List of 3
#>  $ status   : chr "OK"
#>  $ copyright: chr "Copyright (c) 2020 The New York Times Company. All Rights Reserved."
#>  $ response :List of 2
#>   ..$ docs:'data.frame': 10 obs. of  27 variables:
#>   ..$ meta:List of 3
```

That looks intimidating! But it's really just a big, nested list. Let's see what we got in there.

```
# see all items
names(response_df)
#> [1] "status"    "copyright" "response"

# This is boring
response_df$status
#> [1] "OK"
```

```
# So is this
response_df$copyright
#> [1] "Copyright (c) 2020 The New York Times Company. All Rights Reserved."

# This is what we want!
names(response_df$response)
#> [1] "docs" "meta"
```

Within `response_df$response`, we can extract a number of interesting results, including the number of total hits, as well as information on the first ten documents:

```
# What's in 'meta'?
response_df$response$meta
#> $hits
#> [1] 1622621
#>
#> $offset
#> [1] 0
#>
#> $time
#> [1] 403

# pull out number of hits
response_df$response$meta$hits
#> [1] 1622621

# Check out docs
names(response_df$response$docs)
#>  [1] "abstract"               "web_url"
#>  [3] "snippet"                "lead_paragraph"
#>  [5] "print_section"          "print_page"
#>  [7] "source"                 "multimedia"
#>  [9] "keywords"               "pub_date"
#> [11] "document_type"          "news_desk"
#> [13] "section_name"           "type_of_material"
#> [15] "_id"                    "word_count"
#> [17] "uri"                    "headline.main"
#> [19] "headline.kicker"        "headline.content_kicker"
#> [21] "headline.print_headline" "headline.name"
#> [23] "headline.seo"           "headline.sub"
#> [25] "byline.original"        "byline.person"
#> [27] "byline.organization"

# put it in another variable
docs <- response_df$response$docs
```

17.4.3 Iteration through results pager

That's great. But we only have 10 items. The original response said we had 168 hits! Which means we have to make 168 /10, or 17 requests to get them all. Sounds like a job for iteration!

First, let's write a function that passes a search term and a page number, and returns a dataframe of articles.

```
nytapi <- function(term = NULL, n){
    base.url = "http://api.nytimes.com/svc/search/v2/articlesearch.json"
    key = "OnzOBobMTn2IRJ7krcT5RXHknkGLqiaI"

    # Send GET request
    r <- GET(base.url, query = list(`q` = term,
                                    `api-key` = key,
                                    `page` = n))

    # Parse response to JSON
    response <- httr::content(r, "text")
    response_df <- fromJSON(response, simplifyDataFrame = T, flatten = T)

    print(paste("Scraping page: ", as.character(n)))

    return(response_df$response$docs)
}

docs <- nytapi("John Mearsheimer", 2)
#> [1] "Scraping page:  2"
```

Now, we're ready to iterate over each page. First, let's review what've done so far:

```
# set key and base
base.url = "http://api.nytimes.com/svc/search/v2/articlesearch.json"
key = "OnzOBobMTn2IRJ7krcT5RXHknkGLqiaI"
search_term = "John Mearsheimer" #change me

# Send GET request
r <- GET(base.url, query = list(`fq` = search_term,
                                `api-key` = key))

# Parse response to JSON
response <- httr::content(r, "text")
response_df <- fromJSON(response, simplifyDataFrame = T, flatten = T)

# extract hits -- BUGGGGG
```

```
# hits = response_df$response$meta$hits
hits = 168

# get number of pages
pages = ceiling(hits/10)

# modify function to sleep
nytapi_slow <- slowly(nytapi, rate = rate_delay(1))

# iterate over pages, getting all docs
docs_list <- map((1:pages), ~nytapi_slow(term = search_term, n = .))
#> [1] "Scraping page:  1"
#> [1] "Scraping page:  2"
#> [1] "Scraping page:  3"
#> [1] "Scraping page:  4"
#> [1] "Scraping page:  5"
#> [1] "Scraping page:  6"
#> [1] "Scraping page:  7"
#> [1] "Scraping page:  8"
#> [1] "Scraping page:  9"
#> [1] "Scraping page:  10"
#> [1] "Scraping page:  11"
#> [1] "Scraping page:  12"
#> [1] "Scraping page:  13"
#> [1] "Scraping page:  14"
#> [1] "Scraping page:  15"
#> No encoding supplied: defaulting to UTF-8.
#> [1] "Scraping page:  16"
#> No encoding supplied: defaulting to UTF-8.
#> [1] "Scraping page:  17"

# flatten to create one bit dataframe
docs_df <- bind_rows(docs_list)
```

17.4.4 Visualizing Results

To figure out how John Mearsheimer's popularity is changing over time, all we need to do is add an indicator for the year and month each article was published in.

```
# Format pub_date using lubridate
docs_df$date <- ymd_hms(docs_df$pub_date)

by_month <- docs_df %>% group_by(floor_date(date, "month")) %>%
```

```
  summarise(count  = n()) %>%
  rename(month = 1)

by_month %>%
  ggplot(aes(x = month, y = count)) +
  geom_point() +
  theme_bw() +
  xlab(label = "Date") +
  ylab(label = "Article Count") +
  ggtitle(label = "Coverage of John Mearsheimer")
```

17.4.5 More resources

The documentation for httr includes two useful vignettes:

1. httr quickstart guide - summarizes all the basic httr functions like above
2. Best practices for writing an API package - document outlining the key issues involved in writing API wrappers in R

17.5 Webscraping

If no API is available, we can scrape a website directory. Webscraping has a number of benefits and challenges compared to APIs:

Webscraping Benefits

- Any content that can be viewed on a webpage can be scraped. Period
- No API needed
- No rate-limiting or authentication (usually)

Webscraping Challenges

- Rarely tailored for researchers
- Messy, unstructured, inconsistent
- Entirely site-dependent

Some Disclaimers

- Check a site's terms and conditions before scraping.
- Be nice - don't hammer the site's server. Review these ethical webscraping tips
- Sites change their layout all the time. Your scraper will break.

17.5.1 What's a website?

A website is some combination of codebase and database. The "front end" product is HTML + CSS stylesheets + javascript, looking something like this:

Your browser turns that into a nice layout.

Current Senate Members *99th General Assembly*

Leadership Officers Senate Seating Chart **Democrats:** 39 **Republicans:** 20

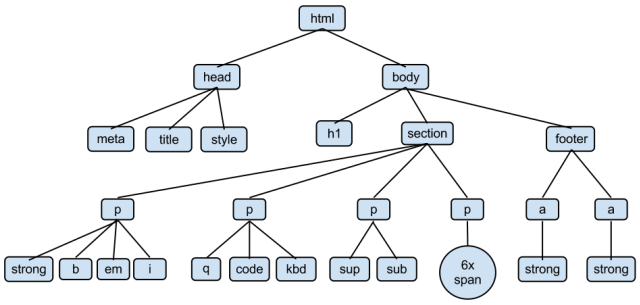
Senator	Bills	Committees	District	Party
Pamela J. Althoff	Bills	Committees	32	R
Neil Anderson	Bills	Committees	36	R
Jason A. Barickman	Bills	Committees	53	R
Scott M. Bennett	Bills	Committees	52	D
Jennifer Bertino-Tarrant	Bills	Committees	49	D
Daniel Biss	Bills	Committees	9	D
Tim Bivins	Bills	Committees	45	R
William E. Brady	Bills	Committees	44	R
Melinda Bush	Bills	Committees	31	D
James F. Clayborne, Jr.	Bills	Committees	57	D
Jacqueline Y. Collins	Bills	Committees	16	D
Michael Connelly	Bills	Committees	21	R
John J. Cullerton	Bills	Committees	6	D

17.5.2 HTML

The core of a website is **HTML** (Hyper Text Markup Language.) HTML is composed of a tree of HTML _nodes**elements**, such as headers, paragraphs, etc.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page title</title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

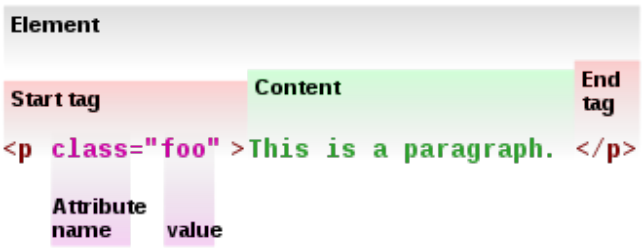
HTML elements can contain other elements:



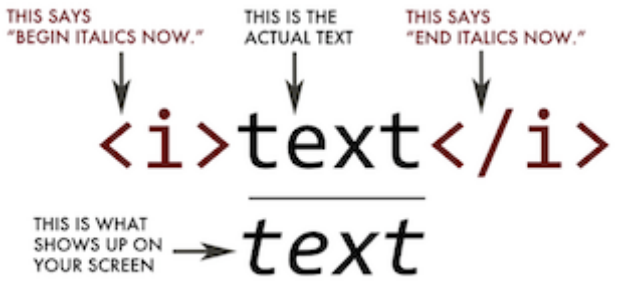
Generally speaking, an HTML element has three components:

1. Tags (starting and ending the element)
2. Atributes (giving information about the element)
3. Text, or Content (the text inside the element)

```
knitr::include_graphics(path = "img/html-element.png")
```



HTML: Tags

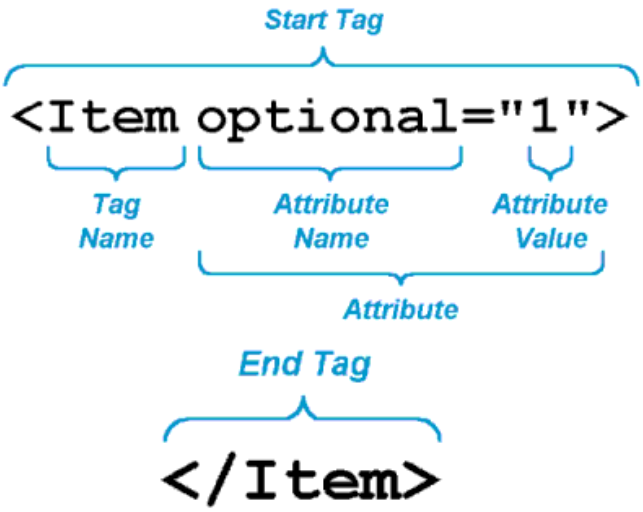


Common HTML tags

Tag	Meaning
`<head>`	page header (metadata, etc
`<body>`	holds all of the content
`<p>`	regular text (paragraph)
`<h1>,<h2>,<h3>`	header text, levels 1, 2, 3
`ol,,<ul>,<li>`	ordered list, unordered list, list item
`<a href="page.html">`	link to "page.html"
`<table>,<tr>,<td>`	table, table row, table item
`<div>,<span`	general containers

HTML Attributes

- HTML elements can have attributes.
- Attributes provide additional information about an element.
- Attributes are always specified in the start tag.
- Attributes come in name/value pairs like: name="value"



- Sometimes we can find the data we want just by using HTML tags or attributes (e.g, all the `<a>` tags)
- More often, this isn't enough: There might be 1000 `<a>` tags on a page. But maybe we want only the `<a>` tags *inside* of a `<p>` tag.
- Enter CSS

17.5.3 CSS

CSS stands for **Cascading Style Sheet**. CSS defines how HTML elements are to be displayed.

HTML came first. But it was only meant to define content, not format it. While HTML contains tags like `<font>` and `<color>`, this is a very inefficient way to develop a website.

To solve this problem, CSS was created specifically to display content on a webpage. Now, one can change the look of an entire website just by changing one file.

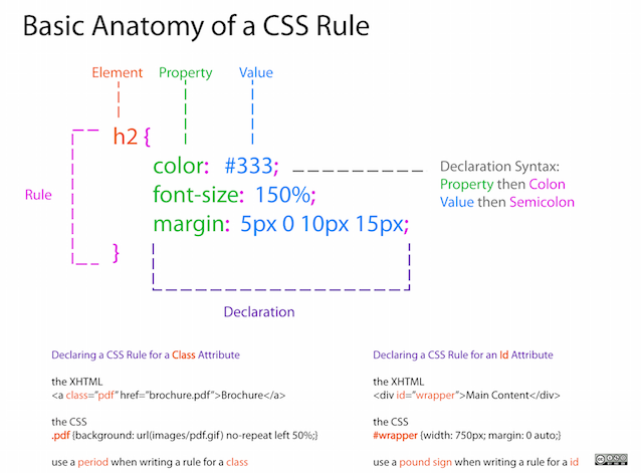
Most web designers litter the HTML markup with tons of `class`es and `id`s to provide "hooks" for their CSS.

You can piggyback on these to jump to the parts of the markup that contain the data you need.

CSS Anatomy

- Selectors
  - Element selector: `p`

  - Class selector: `p class="blue"`
  - I.D. selector: `p id="blue"`
- Declarations
  - Selector: `p`
  - Property: `background-color`
  - Value: `yellow`
- Hooks



17.5.3.1 CSS + HTML

```
<body>
    <table id="content">
        <tr class='name'>
            <td class='firstname'>
                Kurtis
            </td>
            <td class='lastname'>
                McCoy
            </td>
        </tr>
        <tr class='name'>
            <td class='firstname'>
                Leah
            </td>
            <td class='lastname'>
                Guerrero
            </td>
        </tr>
    </table>
```

```
</body>
```

Challenge 1

Find the CSS selectors for the following elements in the HTML above.

(Hint: There will be multiple solutions for each)

1. The entire table
2. The row containing "Kurtis McCoy"
3. Just the element containing first names

17.5.4 Finding Elements with Selector Gadget

Selector Gadget is a browser plugin to help you find HTML elements. Install Selector Gadget on your browser by following these instructions.

Once installed, run Selector Gadget and simply click on the type of information you want to select from the webpage. Once this is selected, you can then click the pieces of information you **don't** want to keep. Do this until only the pieces you want to keep remain highlighted, then copy the selector from the bottom pane.

Here's the basic strategy of webscraping:

1. Use Selector Gadget to see how your data is structured
2. Pay attention to HTML tags and CSS selectors
3. Pray that there is some kind of pattern
4. Use R and add-on modules like `RVest` to extract just that data.

Challenge 2

Go to http://rochelleterman.github.io/. Using Selector Gadget,

1. Find the CSS selector capturing all rows in the table.
2. Find the image source URL.
3. Find the HREF attribute of the link.

17.6 Scraping Presidential Statements

To demonstrate webscraping in R, we're going to collect records on presidential statements here: https://www.presidency.ucsb.edu/

Let's say we're interested in how presidents speak about "space exploration". On the website, we punch in this search term, and we get the following 295 results.

Our goal is to scrape these records, and store pertenant information in a dataframe. We will be doing this in two steps:

1. Write a function to scrape each individual record page (these notes).
2. Use this function to loop through all results, and collect all pages (homework).

Load the following packages to get started:

```
library(tidyverse)
library(rvest)
library(stringr)
library(purrr)
library(knitr)
```

17.6.1 Using `RVest` to Read HTML

The package `RVest` allows us to:

1. Collect the HTML source code of a webpage
2. Read the HTML of the page
3. Select and keep certain elements of the page that are of interest

Let's start with step one. We use the `read_html` function to call the results URL and grab the HTML response. Store this result as an object.

```
document1 <- read_html("https://www.presidency.ucsb.edu/documents/special-message-the-

#Let's take a look at the object we just created
document1
#> {html_document}
#> <html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/content/
#> [1] <head profile="http://www.w3.org/1999/xhtml/vocab">\n<meta charset="utf-8 ...
#> [2] <body class="html not-front not-logged-in one-sidebar sidebar-first page- ...
```

This is pretty messy. We need to use `RVest` to make this information more useable.

17.6.2 Find Page Elements

`RVest` has a number of functions to find information on a page. Like other webscraping tools, RVest lets you find elements by their:

1. HTML tags

2. HTML Attributes
3. CSS Selectors

Let's search first for HTML tags.

The function `html_nodes` searches a parsed HTML object to find all the elements with a particular HTML tag, and returns all of those elements.

What does the example below do?

```
html_nodes(document1, "a")
#> {xml_nodeset (75)}
#>  [1] <a href="#main-content" class="element-invisible element-focusable">Skip ...
#>  [2] <a href="https://www.presidency.ucsb.edu/">The American Presidency Proje ...
#>  [3] <a class="btn btn-default" href="https://www.presidency.ucsb.edu/about"> ...
#>  [4] <a class="btn btn-default" href="/advanced-search"><span class="glyphico ...
#>  [5] <a href="https://www.ucsb.edu/" target="_blank"><img alt="ucsb wordmark  ...
#>  [6] <a href="/documents" class="active-trail dropdown-toggle" data-toggle="d ...
#>  [7] <a href="/documents/presidential-documents-archive-guidebook">Guidebook</a>
#>  [8] <a href="/documents/category-attributes">Category Attributes</a>
#>  [9] <a href="/statistics">Statistics</a>
#> [10] <a href="/media" title="">Media Archive</a>
#> [11] <a href="/presidents" title="">Presidents</a>
#> [12] <a href="/analyses" title="">Analyses</a>
#> [13] <a href="https://giving.ucsb.edu/Funds/Give?id=185" title="">GIVE</a>
#> [14] <a href="/documents/presidential-documents-archive-guidebook" title="">A ...
#> [15] <a href="/documents" title="" class="active-trail">Categories</a>
#> [16] <a href="/documents/category-attributes" title="">Attributes</a>
#> [17] <a href="/documents/app-categories/presidential" title="Presidential (73 ...
#> [18] <a href="/documents/app-categories/spoken-addresses-and-remarks/presiden ...
#> [19] <a href="/documents/app-categories/spoken-addresses-and-remarks/presiden ...
#> [20] <a href="/documents/app-categories/written-presidential-orders/president ...
#> ...
```

That's a lot of results! Many elements on a page will have the same HTML tag. For instance, if you search for everything with the `a` tag, you're likely to get a lot of stuff, much of which you don't want.

In our case, we only want the links corresponding to the speaker Dwight D. Eisenhower.

DWIGHT D. EISENHOWER

34th President of the United States: 1953 - 1961

Special Message to the Congress Relative to Space Science and Exploration.

April 02, 1958

Using selector gadget, we found out that the CSS selector for document's speaker is `.diet-title a`.

We can then modify our argument in `html_nodes` to look for this more specific CSS selector.

```
html_nodes(document1, ".diet-title a")
#> {xml_nodeset (1)}
#> [1] <a href="/people/president/dwight-d-eisenhower">Dwight D. Eisenhower</a>
```

17.6.3 Get Attributes and Text of Elements

Once we identify elements, we want to access information in that element. Oftentimes this means two things:

1) Text
2) Attributes

Getting the text inside an element is pretty straightforward. We can use the `html_text()` command inside of `RVest` to get the text of an element:

```
#Scrape individual document page
document1 <- read_html("https://www.presidency.ucsb.edu/documents/special-message-the-

#identify element with Speaker name
speaker <- html_nodes(document1, ".diet-title a") %>%
  html_text() #select text of element

speaker
#> [1] "Dwight D. Eisenhower"
```

You can access a tag's attributes using `html_attr`. For example, we often want to get a URL from an `a` (link) element. This is the URL is the link "points" to. It's contained in the attribut `href`:

```
speaker_link <- html_nodes(document1, ".diet-title a") %>%
  html_attr("href")
```

```
speaker_link
#> [1] "/people/president/dwight-d-eisenhower"
```

17.6.4 Let's DO this.

Believe it or not, that's all you need to scrape a website. Let's apply these skills to scrape a sample document from the UCSB website – the first item in our search results.

We'll collect the document's date, speaker, title, and full text.

1. Date

```
document1 <- read_html("https://www.presidency.ucsb.edu/documents/special-message-the-congress-re

date <- html_nodes(document1, ".date-display-single") %>%
  html_text() %>% # grab element text
  mdy() #format using lubridate
date
#> [1] "1958-04-02"
```

2. Speaker

```
#Speaker
speaker <- html_nodes(document1, ".diet-title a") %>%
  html_text()
speaker
#> [1] "Dwight D. Eisenhower"
```

3. Title

```
#Title
title <- html_nodes(document1, "h1") %>%
  html_text()
title
#> [1] "Special Message to the Congress Relative to Space Science and Exploration."
```

4. Text

```
#Text
text <- html_nodes(document1, "div.field-docs-content") %>%
          html_text()

#this is a long document, so let's just display the first 1000 characters
text %>% substr(1, 1000)
#> [1] "\n    To the Congress of the United States:\nRecent developments in long-range rockets fo
```

17.6.5 Challenge 1: Make a function

Make a function called `scrape_docs` that accepts a URL of an individual document, scrapes the page, and returns a list containing the document's date, speaker, title, and full text.

This involves:

- Requesting the HTML of the webpage using the full URL and RVest.
- Using RVest to locate all elements on the page we want to save.
- Storing each of these items into a list.
- Returning this list.

```
scrape_docs <- function(URL){

  # YOUR CODE HERE

}

# uncomment to test
# scrape_doc("https://www.presidency.ucsb.edu/documents/letter-t-keith-glennan-adminis
```

Chapter 18

Text Analysis

This unit focuses on computational text analysis (or "text-as-data"). We'll explore:

1. **Preprocessing** a corpus for common text analysis.
2. **Sentiment Analysis and Dictionary Methods**, a simple, supervised method for classification.
3. **Distinctive Words**, or word-separating techniques to compare corpora.
4. **Structural Topic Models**, a popular unsupervised method for text exploration and analysis.

These materials are based off a longer, week-long intensive workshop on computational text analysis. If you're interested in text-as-data, I would encourage you to work through these materials on your own: https://github.com/rochelleterman/FSUtext

18.1 Preprocessing

First let's load our required packages.

```
library(tm) # Framework for text mining
library(tidyverse) # Data preparation and pipes $>$
library(ggplot2) # for plotting word frequencies
library(wordcloud) # wordclouds!
```

A **corpus** is a collection of texts, usually stored electronically, and from which we perform our analysis. A corpus might be a collection of news articles from Reuters or the published works of Shakespeare.

Within each corpus we will have separate articles, stories, volumes, each treated as a separate entity or record. Each unit is called a **document.**

For this unit, we will be using a section of Machiavelli's Prince as our corpus. Since The Prince is a monograph, we have already "chunked" the text, so that each short paragraph or "chunk" is considered a "document."

18.1.1 From Words to Numbers

Corpus Readers

The `tm` package supports a variety of sources and formats. Run the code below to see what it includes

```
getSources()
#> [1] "DataframeSource" "DirSource"       "URISource"       "VectorSource"
#> [5] "XMLSource"       "ZipSource"
getReaders()
#>  [1] "readDataframe"           "readDOC"
#>  [3] "readPDF"                 "readPlain"
#>  [5] "readRCV1"                "readRCV1asPlain"
#>  [7] "readReut21578XML"        "readReut21578XMLasPlain"
#>  [9] "readTagged"              "readXML"
```

Here we'll be reading documents from a csv file. Each row being a document, and columns for text and metadata (information about each document). This is the easiest option if you have metadata.

```
docs.df <-read.csv("data/mach.csv", header=TRUE) #read in CSV file
# docs.df$text <- Encoding("UTF-8")
docs <- Corpus(VectorSource(docs.df$text))
docs
#> <<SimpleCorpus>>
#> Metadata:  corpus specific: 1, document level (indexed): 0
#> Content:  documents: 188
```

Once we have the corpus, we can inspect the documents using inspect()

```
# see the 16th document
inspect(docs[16])
#> <<SimpleCorpus>>
#> Metadata:  corpus specific: 1, document level (indexed): 0
#> Content:  documents: 1
#>
#> [1]  Therefore, since a ruler cannot both practise this virtue of generosity and be
```

Preprocessing functions

Many text analysis applications follow a similar 'recipe' for preprecessing, involving (the order of these steps might differ as per application):

1. Tokenizing the text to unigrams (or bigrams, or trigrams)
2. Converting all characters to lowercase
3. Removing punctuation
4. Removing numbers
5. Removing Stop Words, inclugind custom stop words
6. "Stemming" words, or lemmitization. There are several stemming alogrithms. Porter is the most popular.
7. Creating a Document-Term Matrix

`tm` lets us convert a corpus to a DTM while completing the pre-processing steps in one step.

```
dtm <- DocumentTermMatrix(docs,
           control = list(stopwords = TRUE,
                          tolower = TRUE,
                          removeNumbers = TRUE,
                          removePunctuation = TRUE,
                          stemming=TRUE))
```

Weighting

One common pre-processing step that some applicaitons may call for is applying tf-idf weights. The tf-idf, or term frequency-inverse document frequency, is a weight that ranks the importance of a term in its contextual document corpus. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general. In other words, it places importance on terms frequent in the document but rare in the corpus.

```
dtm.weighted <- DocumentTermMatrix(docs,
           control = list(weighting =function(x) weightTfIdf(x, normalize = TRUE),
                          stopwords = TRUE,
                          tolower = TRUE,
                          removeNumbers = TRUE,
                          removePunctuation = TRUE,
                          stemming=TRUE))
```

Compare first 5 rows and 5 columns of the `dtm` and `dtm.weighted`. What do you notice?

```
inspect(dtm[1:5,1:5])
#> <<DocumentTermMatrix (documents: 5, terms: 5)>>
#> Non-/sparse entries: 3/22
#> Sparsity           : 88%
#> Maximal term length: 7
#> Weighting          : term frequency (tf)
#> Sample             :
#>     Terms
#> Docs abandon abil abject abl ablest
#>    1       0    0      0   0      0
#>    2       0    1      0   0      0
#>    3       0    0      0   0      0
#>    4       0    1      0   1      0
#>    5       0    0      0   0      0
inspect(dtm.weighted[1:5,1:5])
#> <<DocumentTermMatrix (documents: 5, terms: 5)>>
#> Non-/sparse entries: 3/22
#> Sparsity           : 88%
#> Maximal term length: 7
#> Weighting          : term frequency - inverse document frequency (normalized) (tf-i
#> Sample             :
#>     Terms
#> Docs abandon   abil abject    abl ablest
#>    1       0 0.0000      0 0.0000      0
#>    2       0 0.0402      0 0.0000      0
#>    3       0 0.0000      0 0.0000      0
#>    4       0 0.0310      0 0.0228      0
#>    5       0 0.0000      0 0.0000      0
```

18.1.2 Exploring the DTM

Dimensions

Let's look at the structure of our DTM. Print the dimensions of the DTM. How many documents do we have? How many terms?

```
# how many documents? how many terms?
dim(dtm)
#> [1]  188 2368
```

Frequencies

We can obtain the term frequencies as a vector by converting the document term matrix into a matrix and using `colSums` to sum the column counts:

```
 # how many terms?
freq <- colSums(as.matrix(dtm))
freq[1:5]
#> abandon    abil  abject     abl  ablest
#>       4      35       1      61       1
length(freq)
#> [1] 2368
```

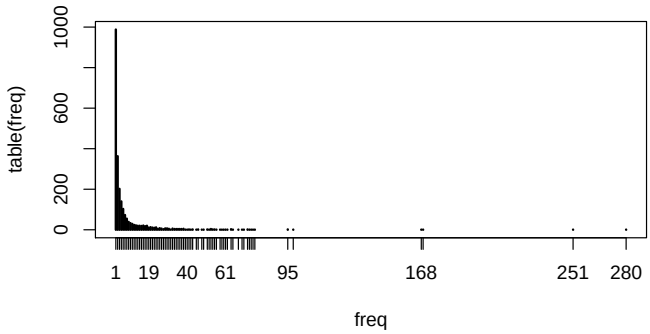
By ordering the frequencies we can list the most frequent terms and the least frequent terms.

```
# order
sorted <- sort(freq, decreasing = T)

# Least frequent terms
head(sorted)
#> ruler  will power   one peopl alway
#>   280   251   169   168    98    95

# most frequent
tail(sorted)
#>  xxiv   xxv  xxvi yield  yoke youth
#>     1     1     1     1     1     1
```

Plotting frequencies

Let's make a plot that shows the frequency of frequencies for the terms. (For example, how many words are used only once? 5 times? 10 times?)

```
# frequency of frenquencies
head(table(freq),15)
#> freq
#>   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
#> 988 363 202 140 103  73  55  39  33  29  24  22  20  20  19
tail(table(freq),15)
#> freq
#>  65  68  70  71  73  74  75  76  77  95  98 168 169 251 280
#>   1   1   1   1   2   1   1   1   1   2   1   1   1   1   1

# plot
plot(table(freq))
```



What does this tell us about the nature of language?

We can reorder columns of DTM to show most frequent terms first:

```
dtm.ordered <- dtm[,order(freq, decreasing = T)]
inspect(dtm.ordered[1:5,1:5])
#> <<DocumentTermMatrix (documents: 5, terms: 5)>>
#> Non-/sparse entries: 10/15
#> Sparsity           : 60%
#> Maximal term length: 5
#> Weighting          : term frequency (tf)
#> Sample             :
#>     Terms
#> Docs one peopl power ruler will
#>    1   0     0     0     1    1
#>    2   3     0     0     1    3
#>    3   0     0     0     0    0
#>    4   0     0     0     1    1
#>    5   3     0     0     1    1
```

Exploring common words

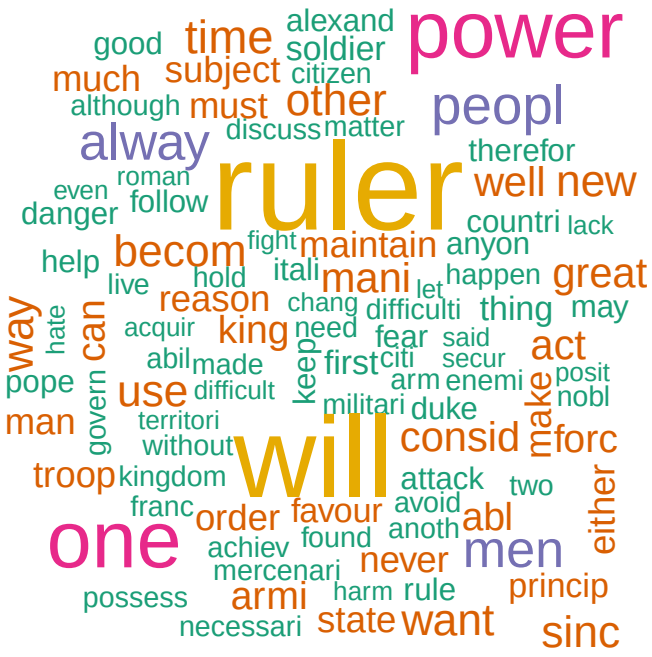
The TM package has lots of useful functions to help you explore common words and associations:

```
# Have a look at common words
findFreqTerms(dtm, lowfreq=50) # words that appear at least 100 times
#>  [1] "abl"      "act"      "alway"    "armi"     "becom"    "can"
#>  [7] "consid"   "either"   "forc"     "great"    "king"     "maintain"
#> [13] "make"     "man"      "mani"     "men"      "much"     "must"
#> [19] "never"    "new"      "one"      "order"    "other"    "peopl"
#> [25] "power"    "reason"   "ruler"    "sinc"     "state"    "subject"
#> [31] "time"     "troop"    "use"      "want"     "way"      "well"
#> [37] "will"
```

```
# Which words correlate with "war"?
findAssocs(dtm, "war", 0.3)
#> $war
#>     wage    fight  antioch     argu     brew    induc      lip    maxim
#>     0.73     0.52     0.45     0.45     0.45     0.45     0.45     0.45
#>  relianc     sage    trifl  postpon     mere     evil    avoid     flee
#>     0.45     0.45     0.45     0.41     0.35     0.34     0.32     0.32
#>   occupi     glad glorious    heard     hunt ineffect     knew   produc
#>     0.32     0.30     0.30     0.30     0.30     0.30     0.30     0.30
#> temporis
#>     0.30
```

We can even make wordclouds showing the most commons terms:

```
# wordcoulds!
set.seed(123)
wordcloud(names(sorted), sorted, max.words=100, colors=brewer.pal(6,"Dark2"))
```



Remove sparse terms

Somtimes we want to remove sparse terms and thus inrease efficency. Look up the help file for the function `removeSparseTerms`. Using this function, create an objected called `dtm.s` that contains only terms with <.9 sparsity (meaning they appear in more than 10% of documents).

```
dtm.s <- removeSparseTerms(dtm,.9)
dtm # 2365 terms
#> <<DocumentTermMatrix (documents: 188, terms: 2368)>>
#> Non-/sparse entries: 11754/433430
#> Sparsity           : 97%
#> Maximal term length: 15
#> Weighting          : term frequency (tf)
dtm.s # 135 terms
#> <<DocumentTermMatrix (documents: 188, terms: 136)>>
#> Non-/sparse entries: 4353/21215
#> Sparsity           : 83%
#> Maximal term length: 12
#> Weighting          : term frequency (tf)
```

18.1.3 Exporting the DTM

We can convert a DTM to a matrix or data.frame in order to write to a csv, add meta data, etc.

First create an object that converts the `dtm` to a dataframe (we first have to convert to matrix, and then to dataframe)

```
# coerce into dataframe
dtm <- as.data.frame(as.matrix(dtm))
names(dtm)[1:10]  # names of documents
#>  [1] "abandon"   "abil"      "abject"    "abl"       "ablest"    "abovement"
#>  [7] "abovenam"  "absolut"   "absorb"    "accept"
```

Now add a column called `doc_section`. For the first 100 rows, the value of this column should be "Section 1". For documents 101-188, the section should be "Section 2".

```
# add fake column for section
dtm$doc_section <- "NA"
dtm$doc_section[1:100] <- "Section 1"
dtm$doc_section[101:188] <- "Section 2"
dtm$doc_section <- as.factor(dtm$doc_section)

# check to see if they're the same number of documents per author
summary(dtm$doc_section)
#> Section 1 Section 2
#>       100        88
```

We can now export the dataframe as a csv.

```
write.csv(dtm, "dtm.csv", row.names = F)
```

18.1.4 Challenges

Using the one of the datasetes in the `text-data` repo, create a document term matrix and create a wordcloud of the most common terms.

```
# download text-data
library("usethis")
use_course("plsc-31101/text-data")

# check out what you just downloaded.
# and unzip a dataset.

# YOUR CODE HERE
```

18.2 Sentiment Analysis and Dictionary Methods

To demonstrate sentiment analysis, we're going to explore lyrics from Taylor Swift songs.

Road the code below to get started.

```
require(tm)
require(tidytext)
require(tidyverse)
require(stringr)
require(textdata)

ts <- read.csv("data/taylor_swift.csv")
getwd()
#> [1] "/Users/rochelleterman/Google Drive/Teaching/20_PLSC_31101/2020"
```

18.2.1 Preprocessing and Setup

First we must preprocess the corpus. Create a document-term matrix from the `lyrics` column of the `ts` data frame. Complete the following preprocessing steps: - convert to lower - remove stop words - remove numbers - remove punctuation.

Think: Why is stemming inappropriate for this application?

```
# preprocess and create DTM
docs <- Corpus(VectorSource(ts$lyrics))

dtm <- DocumentTermMatrix(docs,
```

```
            control = list(tolower = TRUE,
                           removeNumbers = TRUE,
                           removePunctuation = TRUE,
                           stopwords = TRUE
                          ))

dtm <- as.data.frame(as.matrix(dtm))
```

Sentiment dictionaries

We're going to use sentiment dictionaries from the `tidytext` package. Using the `get_sentiments` function, load the "bing" dictionary and store it in an object called `sent`.

```
sent <- get_sentiments("bing")
head(sent)
#> # A tibble: 6 x 2
#>   word       sentiment
#>   <chr>      <chr>
#> 1 2-faces    negative
#> 2 abnormal   negative
#> 3 abolish    negative
#> 4 abominable negative
#> 5 abominably negative
#> 6 abominate  negative
```

We'll now add a column to `sent` called `score`. This column should hold a "1" for positive words and "-1" for negative words.

```
sent$score <- ifelse(sent$sentiment=="positive", 1, -1)
```

18.2.2 Scoring the songs

We're now ready to score each song.

(**NB**: There are probably many ways to program a script that performs this task. If you can think of a more elegant way, go for it!)

First, we'll create a dataframe that holds all the words in our dtm along with their sentiment score.

```
# get all the words in our dtm and put it in a dataframe
words = data.frame(word = colnames(dtm))
head(words)
#>        word
#> 1      back
```

```
#> 2 backroads
#> 3       bed
#> 4   believe
#> 5   beneath
#> 6    beside

# get their sentiment scores
words <- merge(words, sent, all.x = T)
head(words)
#>       word sentiment score
#> 1  abigail      <NA>    NA
#> 2   absent      <NA>    NA
#> 3   absurd  negative    -1
#> 4  accused      <NA>    NA
#> 5      ace      <NA>    NA
#> 6 achilles      <NA>    NA

# replace NAs with 0s
words$score[is.na(words$score)] <- 0
head(words)
#>       word sentiment score
#> 1  abigail      <NA>     0
#> 2   absent      <NA>     0
#> 3   absurd  negative    -1
#> 4  accused      <NA>     0
#> 5      ace      <NA>     0
#> 6 achilles      <NA>     0
```

We can now use matrix algebra (!!) to multiply our dtm by the scoring vector. This will return to us a score for each document (i.e., song).

```
# calculate documents scores with matrix algebra!
scores <- as.matrix(dtm) %*% words$score

# put it in the original documents data frame
ts$sentiment <- scores
```

Which song is happiest? Go listen to the song and see if you agree.

18.2.3 Challenges

Challenge 1

Using the code we wrote above, make a function that accepts 1) a vector of texts, and 2) a sentiment dictionary (i.e. a data frame with words and scores), and returns a vector of sentiment scores for each text.

```
sentiment_score <- function(texts, sent_dict){

  # YOUR CODE HERE

  return(scores)
}

# uncomment to test it out!
# sentiment_score(ts$lyrics, sent)
```

Challenge 2

Using the function you wrote above, find out what the most and least positive Taylor Swift album is.

```
# concatenate to make albums
albums <- ts %>%
  group_by(album) %>%
  summarise(lyrics = str_c(lyrics, collapse = ";"))

# first load the dictionary
afinn <- get_sentiments("afinn")

# then run the function
sentiment_score(albums$lyrics, afinn)

# add to original df
albums$sent <- sentiment_score(albums$lyrics, afinn)
```

18.3 Distinctive Words

This lesson finds distinctive words in the speeches of Obama and Trump.

Run the following code to:

1. Import the corpus
2. Create a DTM

```
require(tm)
require(matrixStats) # for statistics

# import corpus
docs <- Corpus(DirSource("Data/trump_obama"))
```

```
# preprocess and create DTM
dtm <- DocumentTermMatrix(docs,
           control = list(tolower = TRUE,
                          removePunctuation = TRUE,
                          removeNumbers = TRUE,
                          stopwords = TRUE,
                          stemming=TRUE))

# print the dimensions of the DTM
dim(dtm)
#> [1]   11 4094

# take a quick look
inspect(dtm[,100:104])
#> <<DocumentTermMatrix (documents: 11, terms: 5)>>
#> Non-/sparse entries: 14/41
#> Sparsity           : 75%
#> Maximal term length: 11
#> Weighting          : term frequency (tf)
#> Sample             :
#>                 Terms
#> Docs             alien align alik aliv allamerican
#>   Obama_2009.txt     0     0    1    0           0
#>   Obama_2010.txt     0     0    1    1           0
#>   Obama_2011.txt     0     1    0    0           0
#>   Obama_2012.txt     0     0    0    1           0
#>   Obama_2013.txt     0     0    0    0           0
#>   Obama_2014.txt     0     0    0    1           0
#>   Obama_2015.txt     1     0    1    0           0
#>   Trump_2017.txt     0     1    0    0           0
#>   Trump_2018.txt     1     0    0    0           1
#>   Trump_2019.txt     3     0    1    1           0
```

Oftentimes scholars will want to compare different corpora by finding the words (or features) distinctive to each corpora. But finding distinctive words requires a decision about what "distinctive" means. As we will see, there are a variety of definitions that we might use.

18.3.1 Unique usage

The most obvious definition of distinctive is "exclusive". That is, distinctive words are those are found exclusively in texts associated with a single speaker (or group). For example, if Trump uses the word "access" and Obama never does, we should count "access" as distinctive.

Finding words that are exclusive to a group is a simple exercise. All we have to do is sum the usage of each word use across all texts for each speaker, and then look for cases where the sum is zero for one speaker.

```
# turn DTM into dataframe
dtm.m <- as.data.frame(as.matrix(dtm))
dtm.m$that <- NULL # fix weird error with stopwords.

# Subset into 2 dtms for each speaker
obama <- dtm.m[1:8,]
trump <- dtm.m[9:11,]

# Sum word usage counts across all texts
obama <- colSums(obama)
trump <- colSums(trump)

# Put those sums back into a dataframe
df <- data.frame(rbind(obama, trump))
df[ ,1:5]
#>        abandon abess abid abil abject
#> obama       2     1    1    7      0
#> trump       1     0    0    1      1

# Get words where one speaker's usage is 0
solelyobama <- unlist(df[1, trump==0])
solelyobama <- solelyobama[order(solelyobama, decreasing = T)] # order them by frequen
head(solelyobama, 10) # get top 10 words for obama
#>      dont technolog      bank     innov    doesnt   teacher      loan      wont
#>        68        31        30        30        29        26        22        22
#>     debat    climat
#>        21        19

solelytrump <- unlist(df[2, obama==0])
solelytrump <- solelytrump[order(solelytrump, decreasing = T)] # order them by frequen
head(solelytrump, 10) # get top 10 words for trump
#>      isi    agent america.  audienc    megan      it. obamacar     alic
#>        9        8        8        8        8        7        7        6
#>   beauti    elvin
#>        6        6
```

As we can see, these words tend not to be terribly interesting or informative. So we will remove them from our corpus in order to focus on identifying distinctive words that appear in texts associated with every speaker.

```
# subset df with non-zero entries
df <- df[,trump>0 & obama>0]
```

```
# how many words are we left with?
ncol(df)
#> [1] 1525
df[,1:5]
#>        abandon abil abl abort abraham
#> obama        2    7  15     1       1
#> trump        1    1   9     1       1
```

18.3.2 Differences in frequences

Another basic approach to identifying distinctive words is to compare the frequencies at which speakers use a word. If one speaker uses a word often across his or her oeuvre and another barely uses the word at all, the difference in their respective frequencies will be large. We can calculate this quantity the following way:

```
# take the differences in frequences
diffFreq <- obama - trump

# sort the words
diffFreq <- sort(diffFreq, decreasing = T)

# the top obama words
head(diffFreq, 10)
#>     will     year      job     work     make      can american  america
#>      306      217      214      186      177      172      165      155
#>      new    peopl
#>      150      147

# the top trump words
tail(diffFreq, 10)
#>  illeg immigr    isi    usa   hero   ryan border  great  thank   drug
#>     -9     -9     -9     -9    -11    -11    -13    -13    -19    -22
```

18.3.3 Differences in averages

This is a good start. But what if one speaker uses more words *overall*? Instead of using raw frenquncies, a better approach would look at the average *rate* at which speakers use various words.

We can calculate this quantity the following way:

1. Normalize the DTM from counts to proportions
2. Take the difference between one speaker's proportion of a word and another's proportion of the same word.

3. Find the words with the highest absolute difference.

```
# normalize into proportions
rowTotals <- rowSums(df) #create vector with row totals, i.e. total number of words pe
head(rowTotals) # notice that one speaker uses more words than the other
#> obama trump
#> 23021  7432

# change frequencies to proportions
df <- df/rowTotals #change frequencies to proportions
df[,1:5]
#>        abandon     abil      abl    abort  abraham
#> obama 8.69e-05 0.000304 0.000652 4.34e-05 4.34e-05
#> trump 1.35e-04 0.000135 0.001211 1.35e-04 1.35e-04

# get difference in proportions
means.obama <- df[1,]
means.trump <- df[2,]
score <- unlist(means.obama - means.trump)

# find words with highest difference
score <- sort(score, decreasing = T)
head(score,10) # top obama words
#>     job    make    busi     let    need    work    help economi  energi     can
#> 0.00620 0.00541 0.00473 0.00426 0.00419 0.00407 0.00388 0.00378 0.00363 0.00346
tail(score,10) # top trump words
#>   border  tonight   immigr     unit    state     drug     must    great
#> -0.00284 -0.00293 -0.00322 -0.00322 -0.00322 -0.00342 -0.00354 -0.00476
#>    thank american
#> -0.00483 -0.00650
```

This is a start. The problem with this measure is that it tends to highlight differences in very frequent words. For example, this method gives greater attention to a word that occurs 30 times per 1,000 words in Obama and 25 times per 1,000 in Trump than it does to a word that occurs 5 times per 1,000 words in Obama and 0.1 times per 1,000 words in Trump This does not seem right. It seems important to recognize cases when one speaker uses a word frequently and another speaker barely uses it.

As this initial attempt suggests, identifying distinctive words will be a balancing act. When comparing two groups of texts, differences in the rates of frequent words will tend to be large relative to differences in the rates of rarer words. Human language is variable; some words occur more frequently than others regardless of who is writing. We need to find a way of adjusting our definition of distinctive in light of this.

18.3.4 Difference in averages, adjustment

One adjustment that is easy to make is to divide the difference in speakers' average rates by the average rate across all speakers. Since dividing a quantity by a large number will make that quantity smaller, our new distinctiveness score will tend to be lower for words that occur frequently. While this is merely a heuristic, it does move us in the right direction.

```
# get the average rate of all words across all speakers
means.all <- colMeans(df)

# now divide the difference in speakers' rates by the average rate across all speakers
score <- unlist((means.obama - means.trump) / means.all)
score <- sort(score, decreasing = T)
head(score,10) # top obama words
#>    student       cant       idea      money        oil     higher       earn
#>       1.78       1.77       1.70       1.67       1.67       1.66       1.60
#> leadership   research    respons
#>       1.60       1.59       1.58
tail(score,10) # top trump words
#>     drug    grace    death    heart   pillar southern  terribl   unfair
#>    -1.77    -1.80    -1.82    -1.82    -1.84    -1.84    -1.84    -1.84
#>     gang     ryan
#>    -1.87    -1.90
```

18.4 Structural Topic Models

This unit gives a brief overview of the `stm` (structural topic model) package. Please read the vignette for more detail.

Structural topic model is a way to estimate a topic model that includes document-level meta-data. One can then see how topical prevalence changes according to that meta-data.

```
library(stm)
```

The data we'll be using for this unit consists of all articles about women published in the New York Times and Washington Post, 1980-2014. You worked with a subset of this data in your last homework.

Load the dataset. Notice that we have the text of the articles, along with some metadata.

```
# Load Data
women <- read.csv('data/women-full.csv')
names(women)
#>  [1] "BYLINE"              "TEXT.NO.NOUN"        "PUBLICATION"
```

```
#>  [4] "TITLE"                "COUNTRY"              "COUNTRY_FINAL"
#>  [7] "YEAR"                 "UID"                  "COUNTRY_NR"
#> [10] "entities"             "LENGTH"               "COUNTRY_TOP_PERCENT"
#> [13] "COUNTRY_CODE"         "TEXT"                 "DATE"
#> [16] "COUNTRY_MAJOR"        "TYPE"                 "REGION"
#> [19] "SUBJECT"
```

18.4.1 Preprocessing

STM has its own unique preprocessing functions and procedure, which I've coded below. Notice that we're going to use the `TEXT.NO.NOUN` column, which contains all the text of the articles without proper nouns (which I removed earlier).

```
# Pre-process
temp<-textProcessor(documents = women$TEXT.NO.NOUN, metadata = women)
#> Building corpus...
#> Converting to Lower Case...
#> Removing punctuation...
#> Removing stopwords...
#> Removing numbers...
#> Stemming...
#> Creating Output...
meta<-temp$meta
vocab<-temp$vocab
docs<-temp$documents

# prep documents in correct format
out <- prepDocuments(docs, vocab, meta)
#> Removing 19460 of 39403 terms (19460 of 1087166 tokens) due to frequency
#> Your corpus now has 4531 documents, 19943 terms and 1067706 tokens.
docs<-out$documents
vocab<-out$vocab
meta <-out$meta
```

Challenge 1

Read the help file for the `prepDocuments` function. Alter the code above (in 2.1) to keep only words that appear in at least 10 documents.

```
# YOUR CODE HERE
```

18.4.2 Estimate Model

We're now going to estimate a topic model with 15 topics by regressing topical prevalence on region and year covariates.

Running full model takes a **long** time to finish. For that reason, we're going to add an argument `max.em.its` which sets the number of iterations. By keeping it low (15) we'll see a rough estimate of the topics. You can always go back and estimate the model to convergence.

```
model <- stm(docs, vocab, 15, prevalence = ~ REGION + s(YEAR), data = meta, seed = 15, max.em.its
#> Beginning Spectral Initialization
#>   Calculating the gram matrix...
#>   Using only 10000 most frequent terms during initialization...
#>   Finding anchor words...
#>      ..............
#>   Recovering initialization...
#>      ..........................................................................................
#> Initialization complete.
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 1 (approx. per word bound = -7.882)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 2 (approx. per word bound = -7.605, relative change = 3.519e-02)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 3 (approx. per word bound = -7.573, relative change = 4.218e-03)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 4 (approx. per word bound = -7.559, relative change = 1.843e-03)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 5 (approx. per word bound = -7.550, relative change = 1.162e-03)
#> Topic 1: women, show, one, like, design
#>  Topic 2: said, women, polic, report, year
#>  Topic 3: women, team, said, game, world
#>  Topic 4: year, book, first, one, mrs
#>  Topic 5: women, said, percent, femal, will
#>  Topic 6: said, women, one, peopl, woman
#>  Topic 7: women, work, said, year, men
```

```
#>  Topic 8: women, sexual, sex, rape, men
#>  Topic 9: women, said, right, law, govern
#>  Topic 10: said, one, famili, women, peopl
#>  Topic 11: women, film, one, said, woman
#>  Topic 12: said, polit, elect, parti, govern
#>  Topic 13: women, said, cancer, studi, health
#>  Topic 14: women, confer, said, deleg, will
#>  Topic 15: said, women, girl, rape, practic
#> ....................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 6 (approx. per word bound = -7.544, relative change = 8.200e-0
#> ....................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 7 (approx. per word bound = -7.539, relative change = 5.961e-0
#> ....................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 8 (approx. per word bound = -7.536, relative change = 4.491e-0
#> ....................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 9 (approx. per word bound = -7.533, relative change = 3.481e-0
#> ....................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 10 (approx. per word bound = -7.531, relative change = 2.759e-
#> Topic 1: show, design, women, fashion, art
#>  Topic 2: said, women, polic, report, offici
#>  Topic 3: women, team, said, game, world
#>  Topic 4: book, year, life, first, work
#>  Topic 5: women, said, femal, percent, will
#>  Topic 6: said, one, protest, peopl, site
#>  Topic 7: women, work, said, year, percent
#>  Topic 8: women, sexual, rape, sex, men
#>  Topic 9: women, said, right, law, govern
#>  Topic 10: said, one, famili, peopl, day
#>  Topic 11: women, film, one, like, woman
#>  Topic 12: polit, said, elect, parti, govern
#>  Topic 13: women, said, abort, cancer, health
#>  Topic 14: women, confer, said, will, world
#>  Topic 15: said, women, girl, rape, case
#> ....................................................................................
#> Completed E-Step (1 seconds).
```

```
#> Completed M-Step.
#> Completing Iteration 11 (approx. per word bound = -7.530, relative change = 2.214e-04)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 12 (approx. per word bound = -7.528, relative change = 1.804e-04)
#> ..........................................................................................
#> Completed E-Step (1 seconds).
#> Completed M-Step.
#> Completing Iteration 13 (approx. per word bound = -7.527, relative change = 1.496e-04)
#> ..........................................................................................
#> Completed E-Step (2 seconds).
#> Completed M-Step.
#> Completing Iteration 14 (approx. per word bound = -7.526, relative change = 1.265e-04)
#> ..........................................................................................
#> Completed E-Step (1 seconds).
#> Completed M-Step.
#> Model Terminated Before Convergence Reached
```

Let's see what our model came up with! The following tools can be used to evaluate the model.

- `labelTopics` gives the top words for each topic.
- `findThoughts` gives the top documents for each topic (the documents with the highest proportion of each topic)

```
# Top Words
labelTopics(model)
#> Topic 1 Top Words:
#>       Highest Prob: show, design, fashion, women, art, one, like
#>       FREX: coutur, fashion, museum, sculptur, ready--wear, jacket, galleri
#>       Lift: ---inch, -ankl, alexandr, armatur, armhol, art-fair, avant
#>       Score: coutur, art, artist, fashion, museum, exhibit, cloth
#> Topic 2 Top Words:
#>       Highest Prob: said, polic, women, kill, report, offici, govern
#>       FREX: polic, suicid, kill, attack, investig, suspect, arrest
#>       Lift: abducte, charanjit, humanity-soak, male-control, sunil, kalpana, ciudad
#>       Score: polic, rape, kill, said, arrest, attack, investig
#> Topic 3 Top Words:
#>       Highest Prob: women, team, game, said, world, play, olymp
#>       FREX: tournament, championship, olymp, soccer, player, game, medal
#>       Lift: -america, -foot--inch, -hole, -kilomet, -rank, -round, -trump
#>       Score: olymp, championship, tournament, team, player, game, medal
#> Topic 4 Top Words:
#>       Highest Prob: book, year, life, first, write, novel, work
#>       FREX: novel, literari, fiction, book, memoir, novelist, poet
```

```
#>       Lift: buster, calla, goncourt, identical-twin, italian-american, kilcher, klo
#>       Score: novel, book, fiction, literari, poet, writer, write
#> Topic 5 Top Words:
#>       Highest Prob: women, said, femal, percent, militari, will, compani
#>       FREX: combat, board, quota, militari, bank, corpor, infantri
#>       Lift: -combat, cpr, gender-divers, nonexecut, outfitt, r-calif, r-ni
#>       Score: women, militari, infantri, combat, percent, quota, femal
#> Topic 6 Top Words:
#>       Highest Prob: protest, said, one, site, peopl, young, video
#>       FREX: orthodox, internet, web, video, rabbi, prayer, site
#>       Lift: balaclava, grrrl, tehrik-, braveheart, drawbridg, gravesit, guerrilla-s
#>       Score: protest, site, orthodox, video, jewish, rabbi, xxxfx
#> Topic 7 Top Words:
#>       Highest Prob: women, work, said, year, percent, men, ese
#>       FREX: ese, factori, employ, incom, worker, job, market
#>       Lift: flextim, management-track, nec, nontransfer, rabenmutt, chiho, fumiko
#>       Score: ese, percent, compani, work, job, women, factori
#> Topic 8 Top Words:
#>       Highest Prob: women, sexual, sex, rape, men, violenc, said
#>       FREX: harass, sexual, sex, assault, brothel, violenc, behavior
#>       Lift: offenc, tarun, chaud, much-lov, newt, tiresom, sex-rel
#>       Score: rape, sexual, harass, violenc, sex, assault, brothel
#> Topic 9 Top Words:
#>       Highest Prob: women, said, right, law, islam, govern, religi
#>       FREX: islam, religi, veil, constitut, saudi, secular, cleric
#>       Lift: afghan-styl, anglo-, archdeacon, bien-aim, episcopaci, fez, government-
#>       Score: islam, law, women, right, religi, ordin, saudi
#> Topic 10 Top Words:
#>       Highest Prob: said, one, famili, peopl, day, like, home
#>       FREX: villag, room, smile, son, couldnt, recal, sit
#>       Lift: charpoy, jet-black, mitra, schermerhorn, single-famili, tyson, uja-fede
#>       Score: villag, husband, fistula, famili, school, girl, said
#> Topic 11 Top Words:
#>       Highest Prob: women, film, one, like, woman, say, play
#>       FREX: film, theater, movi, charact, actress, documentari, audienc
#>       Lift: clive, fine-tun, kaffir, nushus, shrew, nushu, cadel
#>       Score: film, theater, movi, nushu, play, orchestra, femin
#> Topic 12 Top Words:
#>       Highest Prob: polit, elect, parti, minist, presid, govern, said
#>       FREX: voter, elect, parti, prime, candid, vote, cabinet
#>       Lift: ernesto, pinbal, influence-peddl, information-servic, kakuei, left--cen
#>       Score: elect, parti, vote, minist, voter, polit, candid
#> Topic 13 Top Words:
#>       Highest Prob: women, said, abort, cancer, health, studi, breast
#>       FREX: implant, cancer, breast, pill, virus, patient, estrogen
```

```
#>       Lift: acet, adren, ambulatori, analges, anastrozol, antioxid, ashkenazi
#>       Score: cancer, abort, breast, pill, implant, health, virus
#> Topic 14 Top Words:
#>       Highest Prob: women, said, confer, will, world, organ, right
#>       FREX: deleg, confer, forum, page, peac, nongovernment, ambassador
#>       Lift: -glass, barack, brooklyn-born, expansion, foreclosur, guarantor, holden
#>       Score: deleg, confer, forum, page, palestinian, peac, mrs
#> Topic 15 Top Words:
#>       Highest Prob: said, women, rape, court, case, girl, practic
#>       FREX: mutil, genit, circumcis, asylum, sentenc, judg, tribun
#>       Lift: labia, layli, minora, multifaith, paraleg, salim, strip-search
#>       Score: rape, genit, circumcis, mutil, court, sentenc, prosecutor

# Example Docs
findThoughts(model, texts = meta$TITLE, n=2,topics = 1:15)
#> Warning in findThoughts(model, texts = meta$TITLE, n = 2, topics = 1:15): texts
#> are of type 'factor.' Converting to character vectors. Use 'as.character' to
#> avoid this warning in the future.
#>
#>  Topic 1:
#>       KENZO'S CAREFREE STYLES AT AN OFFBEAT SHOWING
#>      A MODERN LOOK, A CLASSIC TOUCH FROM SAINT LAURENT
#>  Topic 2:
#>       Assailants Kill 4 Iraqi Women Working for U.S.; Gunmen Follow Van Carrying Laundry Emplo
#>      WORLD IN BRIEF
#>  Topic 3:
#>       AMERICANS LEAD EAST GERMANS IN TRACK
#>      Russians Chart a New Path
#>  Topic 4:
#>       BEST SELLERS: September 6, 1998
#>      BEST SELLERS: September 13, 1998
#>  Topic 5:
#>       In Britain, a Big Push for More Women to Serve on Corporate Boards
#>      Poll: Allow women in combat units
#>  Topic 6:
#>       Neda's Legacy; A woman's death moves Iranian protesters.
#>      Jewish Feminists Prompt Protests at Wailing Wall
#>  Topic 7:
#>       China Scrambles for Stability as Its Workers Age
#>      A high price for a paycheck; Caught between the demands of the corporate workplace and of
#>  Topic 8:
#>       Confronting Rape in India, and Around the World
#>      Sexual Harassment Prosecutions Get Short Shrift in India, Lawyer Says
#>  Topic 9:
#>       English Church Advances Bid For Women As Bishops
```

```
#>      Egypt Passes Law On Women's Rights; Polygamy Still Allowed for Men
#>  Topic 10:
#>       An Old Cinema in Pakistan Has New Life After Quake
#>      Maria Duran's Endless Wait
#>  Topic 11:
#>       For France, An All-Purpose Heartthrob
#>      Film: Brazilian 'Vera'
#>  Topic 12:
#>       The Widow Of Ex-Leader Wins Race In Panama
#>      Cabinet Defeated in Iceland as Feminists Gain
#>  Topic 13:
#>       SECTION: HEALTH; Pg. T18
#>      Dense Breasts May Need Sonograms to Detect Cancer
#>  Topic 14:
#>       DISPUTES ON KEY ISSUES STALL KENYA PARLEY
#>      'CHAOTIC' CONDITIONS FEARED AT U.N.'S PARLEY ON WOMEN
#>  Topic 15:
#>       Woman Fleeing Tribal Rite Gains Asylum; Genital Mutilation Is Ruled Persecuti
#>      Refugee From Mutilation
```

18.4.2.1 Challenge 2

Estimate other models using 5 and 40 topics, respectively. Look at the top words for each topic. How do the topics vary when you change the number of topics?

Now look at your neighbor's model. Did you get the same results? Why or why not?

```
# YOUR CODE HERE
```

18.4.3 Interprete Model

Let's all load a fully-estimated model that I ran before class.

```
# load the already-estimated model.
load("data/stm.RData")
```

Challenge 3

Using the functions `labelTopics` and `findThoughts`, hand label the 15 topics. Hold these labels as a character vector called `labels`

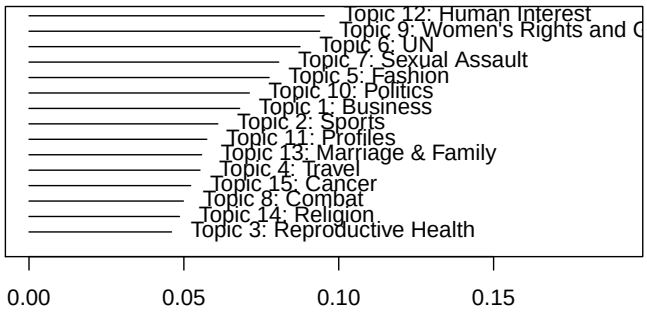
```
# Store your hand labels below.
labels = c()
```

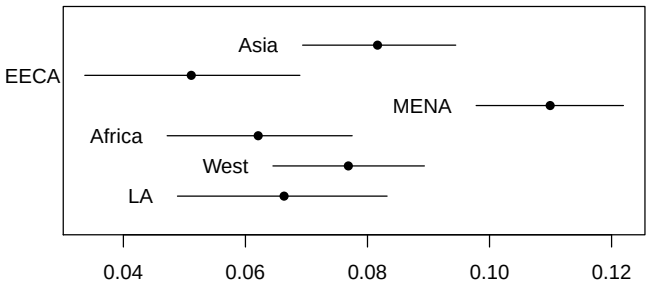
Now look at your neighbor's labels. Did you get the same results? Why or why not?

18.4.4 Analyze topics

We're now going to see how the topics compare in terms of their prevalence across region. What do you notice about the distribution of topic 9?

```
# Corpus Summary
plot.STM(model, type="summary", custom.labels = labels, main="")

# Estimate Covariate Effects
prep <- estimateEffect(1:15 ~ REGION + s(YEAR), model, meta = meta, uncertainty = "Global", docum

# plot topic 9 over region
regions = c("Asia", "EECA", "MENA", "Africa", "West", "LA")
plot.estimateEffect(prep, "REGION", method = "pointestimate", topics = 9, printlegend = TRUE, lab
```





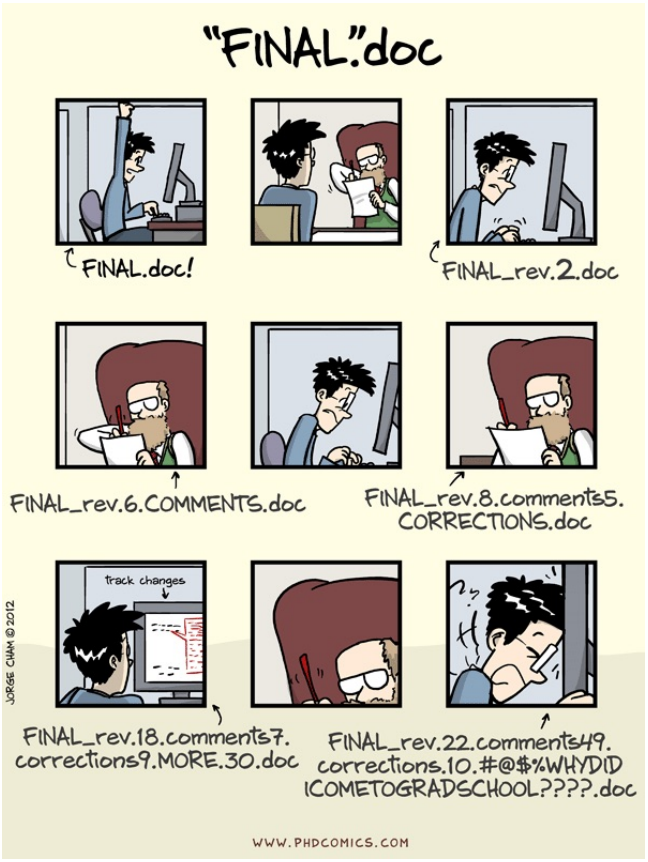
Chapter 19

Git and Github

19.1 Learning Objectives

- Explain which initialization and configuration steps are required once per machine, and which are required once per repository.
- Go through the modify-add-commit cycle for single and multiple files and explain where information is stored at each stage.
- Configure Git to ignore specific files, and explain why it is sometimes useful to do so.
- Explain collaboration options on GitHub
- Go through the fork & pull workflow

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, version control is much better for this than this:



Git is powerful and complicated. We can do a full day workshop on git alone. But it is also quite possible to harness it's powers by cycing through three commands: `add`, `commit`, `push`. So even if you don't understand what's going on underneath the hood, knowing just these three commands can get you very far.

19.2 Starting with Git

The first time we use Git on a new machine, we need to configure a few things.

Here's how Dracula sets up his new laptop:

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

(Please use your own name and email address instead of Dracula's, and please use the same email as you used to make your GitHub account.)

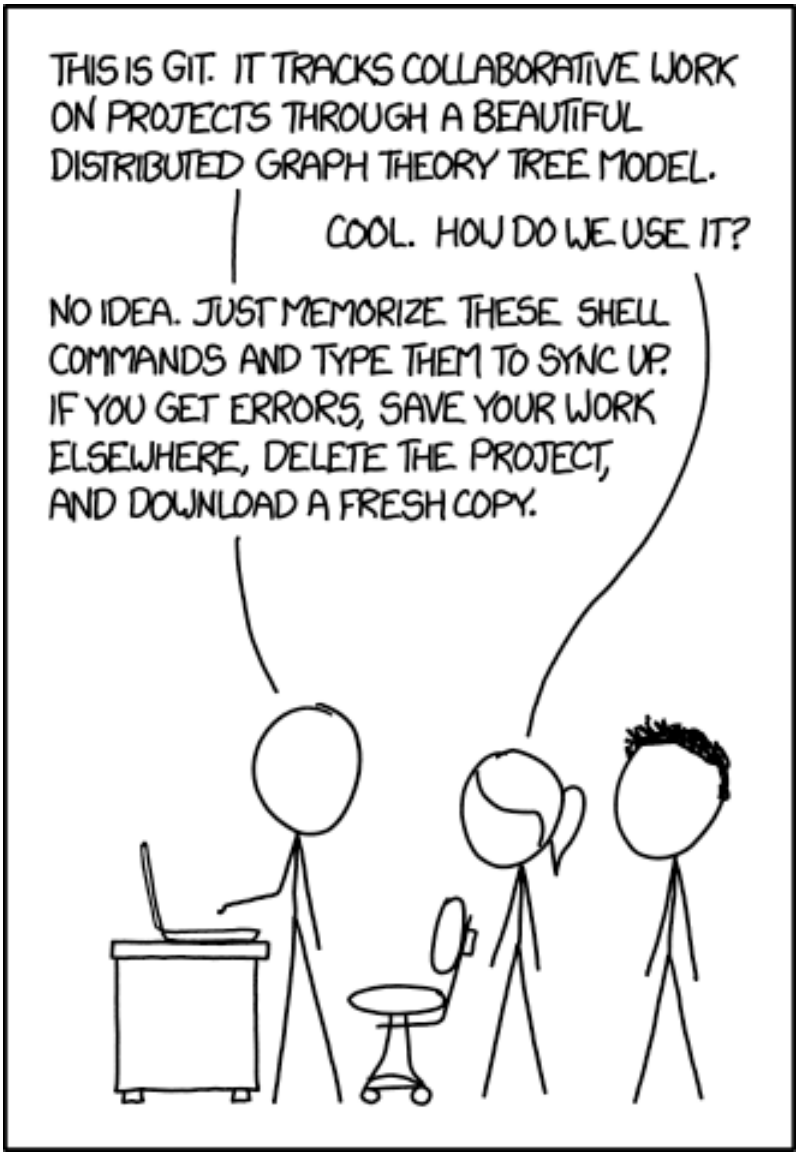


Figure 19.1: xkcd

Git commands are written `git verb`, where `verb` is what we actually want it to do. In this case, we're telling Git to configure our name and email address,

The two commands above only need to be run once: the flag `--global` tells Git to use the settings for every project on this machine.

19.2.1 Creating a repository

Once Git is configured, we can start using it to share code on GitHub.

Follow these instructions to create a new GitHub **repository**. Be sure to add these options:

1. Call the repo "plsc31101-final-project"
2. Create a README.md file
3. Create a .gitignore file
4. Don't add a license for now. Later, you can add a license for your project (see here for information on which license to choose.)

   Git without GitHub

   Git is often used in conjunction with GitHub. But you can also use git to track changes locally on your computer.

   If you wanted to start using Git from scratch on a new project, you can create a directory and tell Git to make it a repository – a place where Git can store old versions of our files – using the command `git init`

   ```
   $ git init
   ```

After you create your directory, **clone** a local copy onto your computer by following these instructions. Be sure to clone in a location that you will remember!

```
$ cd ~
$ git clone https://github.com/vlad/plsc31101-final-project.git
```

Now, navigate into your new git repository

```
$ cd ~/plsc31101-final-project
```

If we use `ls -a` to show the directory's contents, we can see a hidden directory called `.git`:

```
$ ls -a

# .                   .gitignore
# ..                  README.md
# .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
```

19.2.2 `git add`: tracks files

Let's add a file into our directory.

```
$ touch file.txt
```

Now, when we type in `git status`, we see something like this:

```
$ git status

# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
```

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

To add new files, you can either type `git add [file name]` like so:

```
$ git add file.txt
```

OR, if you want to add ALL the new files in a repository, you can use the `.` shortcut:

```
$ git add .
```

Now if we use `git status` we should no longer see any untracked files.

19.2.3 `git commit`: saves files

Git now knows that it's supposed to keep track of all the files in your repo,but it hasn't yet recorded any changes you've made to those files. To get it to do that, we need to run one more command:

```
$ git commit -am "First Commit"

# [master (root-commit) f22b25e] First Commit
 # 1 file changed, 1 insertion(+)
 # create mode 100644 ...
```

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 **revision** and its short identifier is `f22b25e`. (Your revision may have another identifier.)

We use the `-a` flag (for 'all') to tell Git that we want to commit all the changes we've made to every file. If we just run the `git commit` without the `-a` option, Git will expect us to specify which file's changes we want saved.

We use the `-m` flag (for "message") to record a 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 at the start) so that we can write a longer message.

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

    # First commit
```

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

19.2.4 `git push`: moves changes from one branch to another.

Systems like git allow us to move work between any two repositories. In practice, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop.

This is where GitHub comes in: it holds the **master** copy of a repository, and allows us to move changes between multiple **local** copies.

To copy our changes from our laptop to our GitHub repo, we can use `git push`:

```
$ 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/plsc31101-final-project
 # * [new branch]      master -> master
# Branch master set up to track remote branch master from origin.
```

This tells git to push our changes to the repository's "origin" – i.e., the copy on GitHub.

Now open up a web browser and navigate to your GitHub repository. What do you see?

19.2.5 Challenge 1

Navigate to https://github.com/plsc-31101/replication-template and clone the repository to your computer.

Copy all the files and directories in this folder into your new github repo (plsc31101-final-project).

Then add, commit, and push. Use this template for your final project!

Cheat sheet

```
$ git add .
$ git commit -am "commit message"
$ git push origin/master
```

19.2.6 Ignoring Things

Oftentimes we'll have files that we do not want git to track for us. These include sensitive data files, as well as hidden files with extensions like .Rhistory, .ipynb_checkpoints, and .DS_Store (Dropbox).

Let's create a few dummy files:

```
$ touch a.dat b.dat data/c.csv data/d.csv
```

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
#   data/
# 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
data/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `data` 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 recently-changed `.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 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 files to the repository that we don't want.

```
$ 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.
# fatal: no files added
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. 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
```

19.2.7 Challenge 2

Continue editing the .gitignore file to add extentions you don't want to track, like .DS_Store, .Rhistory, etc.

19.2.8 Pulling / Syncing

Oftentimes we need to sync our local repo with the *master branch* (the default branch) on GitHub. For instance, let's say you have two computers, one at home and one at work. We made a change using our work computer, and pushed it to the master branch on GitHub. But then we go home and find that our local copy is out of date.

A more common method of syncing branches is to use `git fetch` followed by `git merge`; or `git pull`.

1. `git fetch` followed by `git merge` *combines* your local changes with changes made by others.
2. `git pull` is a convenient shortcut for completing both `git fetch` and `git merge` in the same command

```
$ git pull origin/master
```

This is helpful if you want to merge your changes and the master branch.

> **Commit before your pull**
>
> Because `pull` performs a merge on the retrieved changes, you should ensure that your local work is committed before running the pull command. If you run into a merge **conflict** you cannot resolve, or if you decide to quit the merge, you can use `git merge --abort` to take the branch back to where it was in before you pulled. See here for more info on fetching and merging.

Sometimes `git pull` will cause **merge conflicts**, meaning that your local repository and master branch diverged on some lines of code, and git doesn't know which version you want to keep.

If you *know* that you want to keep the master branch version, you can overwrite your local repository like this:

```
$ git fetch
$ git reset --hard origin/master
```

Let's break this down:

1. `git fetch` retrieves a record of all changes made in the master branch.
2. `git reset --hard origin/master` will reset your local repo to match the master branch.

With these commands, every tracked file will be overwritten to match to its version in master. Be careful with this: all local changes will be lost.

19.3 Collaborating

Version control really comes into its own when we begin to collaborate with other people.

All of the course notes are contained in their own Github Repo: https://github.com/plsc-31101/course

I've created a directory in this repo called `98-Final-Projects`. We're going to collaborate on this directory using git to collect information about your final projects.

There are two main ways to collaborate on github:

1. Adding individual collaborators to a project
2. The **fork & pull** model.

The first method adds users to your project, giving them full permissions to make changes. For this course, I added Pete Cuppernull as a collaborator, so that he could push commits easily to the repository without my expressed approval.

Collaborating in this fashion is very similar to the workflow described above.

19.3.1 Fork & Pull Model

GitHub also allows you to to accept individual contributions from users without granting them full access. This if referred to as the *Fork & Pull model*.

Fork & Pull involves the following steps:

1. Fork an existing repo

The first step in in this workflow is to **fork** an existing repository. A fork is a copy of a repository that you manage yourself. Forks let you make changes to a project without affecting the original repository.

To fork a repo:

1. On GitHub, navigate to plsc-31101/course,
2. In the top-right corner of the page, click **Fork**.

Now you have a fork of the original repo in *your-user-name*/course

2. Commit a change

We've already seen how you can commit a change directly in GitHub's web interface. But when working with code, you often want to develop your scripts on your computer, so you can test it using R, Python, etc.

To do this, you first need to **clone your fork** onto your computer.

1. On GitHub, navigate to your fork of the `course` repository.
2. In the right sidebar of your fork's repository page, copy the clone URL for your fork.
3. Use `git clone` to clone the repo.

```
$ cd ~
$ git clone https://github.com/YOUR-USERNAME/course
```

We're now ready to make a change to the repo. Create a file in `98-Final-Projects` directory named after yourself. **Protip**: the `touch` command quickly creates an empty file.

```
$ cd course
$ touch 98-Final-Projects/rochelle-terman.md
```

Open up that file in a text editor (I like Sublime Text) and add:

1. The title of your project (in a header 2)
2. A 1-2 sentence description of the project
3. A link to your github repo (that you just made)

   Markdown Reminder

   Files with the extension .md are called **markdown** files. Markdown is a markup language used to convert plain text to HTML and many other formats. It's basically a way to add markup to a text (making things bold, lists, links, etc) using very simple syntax. It is often used in README files in software packages. You may have also noticed that all of the lessons for this course are written in markdown, as is many of the text files on GitHub. You can learn more about how to write GitHub-flavored markdown here.

Then add, commit, and push the change.

```
$ git status
$ git add 16_final-projects/rochelle-terman.md
$ git commit -am "my final project info"
$ git push
```

3. Submit a pull request

Navigate to your GitHub repo (online) and check out your change!

Remember when you forked the repository originally? That means that your repository is different from mine, and from everybody elses. What if you want to share your change with others?

To do this, navigate to your GitHub repository and click the green icon to submit a **pull request**.

After you submit, I have the option to accept.

4. Keep your fork synced

It's good practice to keep your fork synced with the upstream (i.e. the original) repository. That way, if I make a change to PS239T, you can easily pull that change into your fork.

You can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

```
$ git remote -v
$ git remote add upstream https://github.com/plsc-31101/course.git
$ git remove -v
```

With `git remote -v`, you'll see the current configured remote repository for your fork.

Now you can sync your fork with the upsteam repo using `git fetch`:

```
$ git fetch upstream
$ git merge upstream/master
```

To learn more:

1. [GitHub documentation](https://help.github.com
2. This great stackoverflow answer