PLSC 31101: Computational Tools for Social Science

Rochelle Terman

Fall 2020

Contents

Part I

Before Class

Chapter 1

About

This book contains course notes and other materials for PLSC 31101: Introduction to Computational Tools for Social Science. The class was created by Rochelle Terman at the University of Chicago, with contributions from Pete Cuppernull, Jenna Gibson, and Giacomo Ramos.

To Use

Download the materials on your computer by running the following code in RStudio. Note that for this to work, you will need to have `tidyverse` installed.

```
# Install tidyverse if you have not already done so.
install.packages("tidyverse")

# load "usethis" library
library("usethis")

# download course materials
use_course("plsc-31101/course")
```

Acknowledgments

Some of these materials have been adapted from other sources, and we thank them greatly. In alphabetical order:

- Advanced R by Hadley Wickham.
- A ModernDive into R and the tidyverse.
- Computing for Social Sciences.
- Code and Data for the Social Sciences: A Practitioner's Guide. by Gentzkow, Matthew and Jesse M. Shapiro.

- Justin Grimmer.
- R bootcamp at UC Berkeley.
- R for Data Science by Hadley Wickham.
- R Studio.
- Software Carpentry.
- Stat545.

Chapter 2

Syllabus

- Instructor: Rochelle Terman, rterman@uchicago.edu
- TA: Pete Cuppernull, pcuppernull@uchicago.edu
- Time: Mondays and Wednesdays, 1:50 pm – 3:10 pm
- Lab: Wednesdays, 04:10 pm - 05:30 pm
- Zoom link: https://uchicago.zoom.us/j/92579198152?pwd=SzJPSzFyNVpIQm14ZlhJQnJrdDhMUT09
- Office hours:
  - Rochelle Terman: Fridays, 10:00am - 12:00pm (sign up here.)
  - Pete Cuppernull: Tuesdays, 3:00 pm - 5:00pm

2.1 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 with 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.

2.2 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.

2.3 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

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. It is not acceptable to submit homework as a group or to turn in copies of the same code or output. While you are encouraged to use the internet to help you debug, 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 by writing 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 on the Piazza site.
- Collaborating with the computing community 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 did not 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 6** (no more than 2 paragraphs) in order to get approval/feedback from the instructors.

Project materials (i.e., a GitHub repository) will be due by end of day on **December 10**. We will specify submission details in class.

On **December 11 (1:30 pm - 3:30 pm)**, 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, possibly, for the entire course.

2.4 Activities and Materials

Class Format and Zoom

All classes and discussion sections will be held remotely on Zoom at this link: https://uchicago.zoom.us/j/92579198152?pwd=SzJPSzFyNVpIQm14ZlhJQnJrdDhMUT09

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.

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 Notes and Code

All materials will be available on GitHub, including class notes, code demonstrations, sample data, etc.

Download the materials on your computer by running the following code in RStudio. Note that for this to work, you will need to have `tidyverse` installed.

```
# Install tidyverse if you have not already done so.
install.packages("tidyverse")

# load "usethis" library
library("usethis")

# download course materials
use_course("plsc-31101/course")
```

Those materials are also available on this website: https://plsc-31101.github.io/course/. Students will be assigned readings from these notes before every class.

Canvas and Piazza

We will use Canvas for turning in assignments.

We will use Piazza for communication (announcements and questions). You should ask questions about class materials and assignments through the Piazza 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.

Find our Piazza class signup link at: http://piazza.com/uchicago/fall2020/plsc31101

Tech Requirements and Software

See the Install Page page for detailed information on the software we will be using. Please download and install the required software before the first class.

We will be having an **InstallFest on Wednesday, September 30 from 9:30 am to 11:30 am, on Zoom** for those students experiencing difficulties downloading and installing the requisite software.

If you have difficulties installing, please post a question on Piazza with details on what you are trying to install, what actions you took, any error messages, etc.

2.5 Curriculum Outline/Schedule

Week 1: Intro

- M (No Class).
- W 9/30: Intro, R Tools.
- L 9/30: R Markdown + Homework.

Week 2: Working with Data

- M 10/5: R Syntax/coding basics.
- W 10/7: Introduction to data, data transformation with `dplyr`.
- L 10/7: Workflow (scripts, projects, paths, import/export).

Week 3: Data Munging

- M 10/12: Tidying data with `tidyr`.
- W 10/14: Relational data and joins.

- L 10/14:

Week 4: Visualization

- M 10/19: `ggplot`.
- W 10/21: Factors and models.
- L 10/21:

Week 5: R Objects and Indexing

- M 10/26: Vectors.
- W 10/28: Lists and dataframes.
- L 10/28:

Week 6: Strings and Dates

- M 11/2: Strings.
- W 11/4: Regex.
- L 11/4: Dates and times.

Week 7: Programming in R

- M 11/9: Functions and conditionals.
- W 11/11: Iteration.
- L 11/11:

Week 8: Data From the Web

- M 11/16: APIs.
- W 11/18: Webscraping.
- L 11/18: Twitter API.

Week 9: Thanksgiving Break – No Class

Week 10: Text Analysis.

- M 11/30: Text analysis 1.
- W 12/2: Text analysis 2.
- L 12/2:

Week 11: Finals

- FRIDAY 12/11, 1:30 PM-3:30 PM

Chapter 3

Installation

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

3.1 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). Please install **R version 4.0.2 (2020-06-22) – "Taking Off Again"**.

If you use a Mac, this will require at least 10.11 (El Capitan) as your OS. If you use a PC, anything Windows 7 or later will be sufficient.

3.2 RStudio

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

3.3 R Packages

You will 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.

To install packages, open RStudio. Go to Tools > Install Packages. Enter the following: `tidyverse, knitr, gapminder, rtweet, kableExtra, devtools, tm, wordcloud, matrixStats, SnowballC, tidytext, textdata, stm, readtext`

If you have problems installing packages, make sure that you are connected to the internet and that https://cloud.r-project.org/ is not blocked by your firewall or proxy. If RStudio returns an error message, go to "Preferences" and check the "Packages" section. Under "CRAN Mirror," if no mirror is selected, choose "Global (CDN) - RStudio".

3.4 LaTeX

In order to knit R Markdown 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()
```

3.5 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.

Download Git here: https://git-scm.com/downloads.

After installing Git, there will not be anything in your `/Applications` folder, as Git is a command line program.

3.6 Google Chrome

You will need to have a modern web browser installed to perform some of the tasks in this class. The recommended browser for this class is Google Chrome.

3.7 Selector Gadget

As part of the webscraping process, you will need to examine HTML elements in your data. In this class we will be using Selector Gadget for this purpose.

If using Google Chrome, you can simply install the Selector Gadget Chrome extension. If for any reason you cannot use Chrome extensions (including on Chrome itself), you can instead install Selector Gadget by following the instructions on the Selector Gadget website.

3.8 Other Helpful Tools

While not required, I recommend you install Sublime Text, which is a free, advanced text editor.

3.9 Troubleshooting

If you have trouble with installation, please come to the Installfest on **Wednesday, September 30 from 9:30 am to 11:30 am, on Zoom**.

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

If you still have difficulties installing, please post a question on Piazza with details on what you are trying to install, what actions you took, any error messages, etc.

Chapter 4

Homework Rubric

Each question will be graded along the following criteria:

Check Minus: The code does not 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 code.
2) Repetitive/redundant/overly complex code.
3) Graphs and tables lack labels or are otherwise difficult to understand.

Check Plus: The code works and exhibits:

1) Well-documented and organized code.
2) Clean and efficient code.
3) Graphs and tables that have proper labels and are easily readable.

Part II

Course Notes

Chapter 5

Introduction

5.1 The Motivation

Here is 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 is 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 is simply not space to add more about computing without dropping something else.
2. **The blind leading the blind**: The infrastructure does not exist to help scholars develop the skills they need. Senior researchers cannot teach the next generation how to do things that they do not 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, once you get the hang of it, learning how to program can save you an enormous amount of time doing basic tasks that you would otherwise do by hand.

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

A Practical Example

To illustrate, here is 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 metadata for issue, action, and 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**

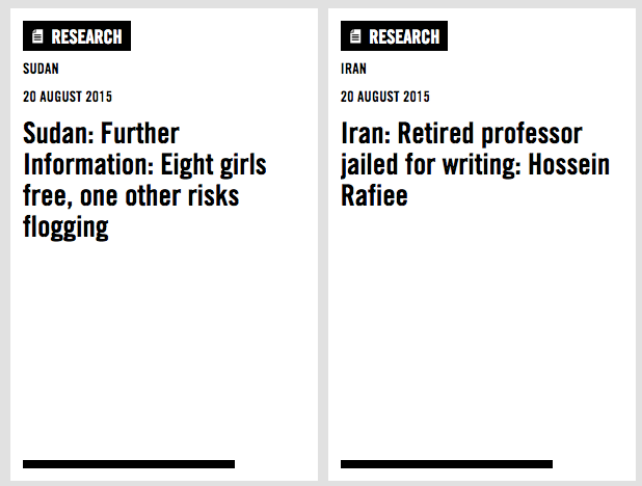
Task 2 (by hand)

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 and add metadata for issue and 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, and sentiment using computational text analysis tools.
3. Uses webscraping to collect all of Amnesty International's urgent actions.
4. Runs simple regression models with R to correlate Amnesty reports with UPR shaming.

Total time: 2 months

Time Saved: 1.5 years

5.2 About This Class

About Me

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

- A few years ago, I did not 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 will not 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 are using R because it is 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 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.

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

Course Websites

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

We will use Canvas to distribute/accept assignments.

We will use Piazza for communications and discussion. Please use Piazza 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 are due the following week.
- Exceptions will be noted.
- The first assignment is due next week before class on Wednesday, October 7. It is available on Canvas.

- Turn in assignments on Canvas.
- Work in groups, but submit your own assignment.

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 Piazza site.
- Collaborating with the computing 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.
2. **Tutorial project**: Create a tutorial on a tool we did not cover in class.

Both options require an R Markdown file narrating the project.

Students are required to write a short proposal by **November 6** (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 10**. We will specify submission details in class.

On **December 11**, we will have a **lightning talk session** where students can present their projects in a maximum 5-minute talk.

Software

- Installation instructions are on the website.
- Get started **EARLY**.
- Go to the Installfest (Wednesday, Sep 30, 9:30-11:30 on Zoom) to double check your installation.
- If you have computer troubles, post the problem on Piazza with as much detail as possible.

5.3 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 language.
4. **Object-oriented**: "Objects" are data + functions. Some programming languages are object-oriented (e.g., Python), and some are not (e.g., C).
5. **Many more**: Here is 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 is 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 is fun, easy, and multi-purpose. Here is an article that can offer more advice.

In this class, we will be using R because it is the most popular language in our disciplinary 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 is 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. **Do not learn specific programming languages; learn how to program.**

What programming is really like.

Here is the sad reality: When you are 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 apply just 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 are a good programmer, you are 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 is better to type than to copy and paste.
6. Most "programming" is actually researching, experimenting, and 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 friends.

Using the Internet

You should also make use of Google and StackOverflow to resolve the error. Here are 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 do not produce the error. Try other people's code, but do not fall into the copy-paste trap.

Asking for Help

We will use 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 are 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 6

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.

6.1 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.

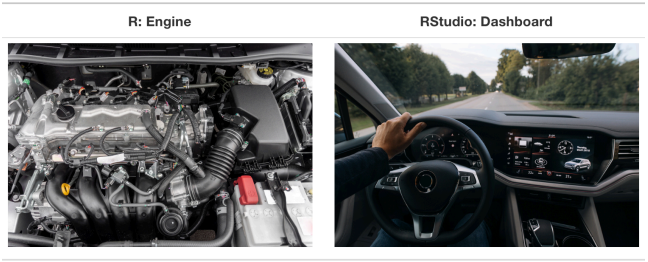
- It has powerful tools for communicating your results. R packages make it easy to produce HTML or PDF reports, or create interactive websites.
- A large and growing community of peers.

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.

6.2 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.

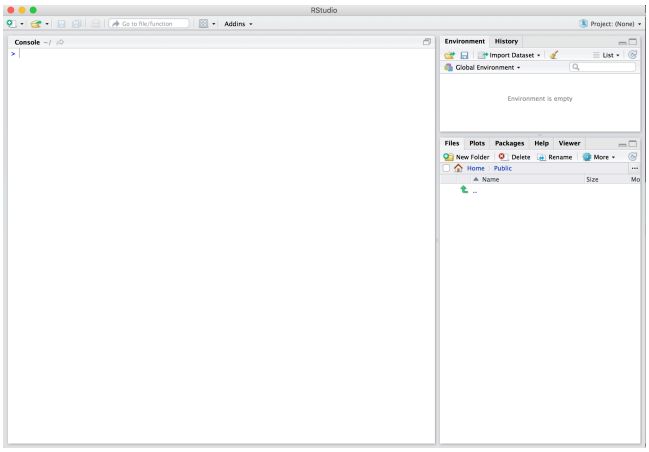
RStudio includes a console, a syntax-highlighting code editor, as well as tools for plotting, history, debuggingm and workspace management. It is also free and open-source. Yay!

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.

R: Do not open this	RStudio: Open this
	

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



6.2.1 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.

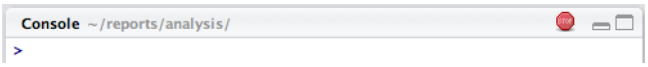
```
> "This is an incomplete quote
+
```

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.

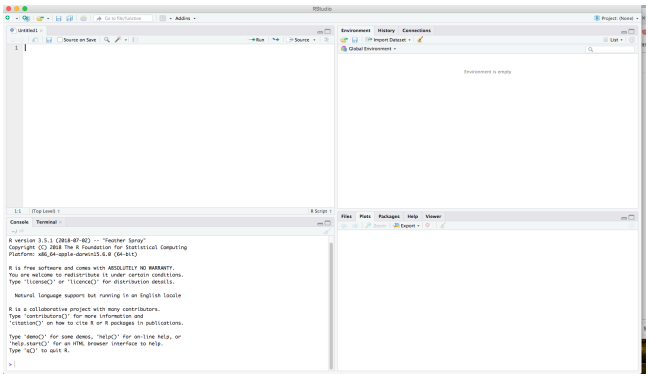
2. Console title bar: This screenshot illustrates a few additional capabilities provided by the console title bar:

   - 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).



6.2.2 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.

6.2.3 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 + Enter`. Doing this regularly is a great way to check that you have captured all the important parts of your code in the script.

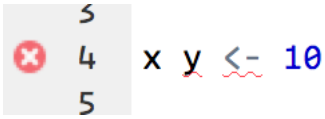
6.2.4 Comments

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 #.

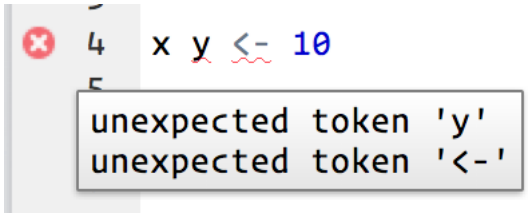
```
# This is a comment.
# This is another line of comments.
```

6.2.5 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.

6.2.6 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!*

6.2.7 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)
```

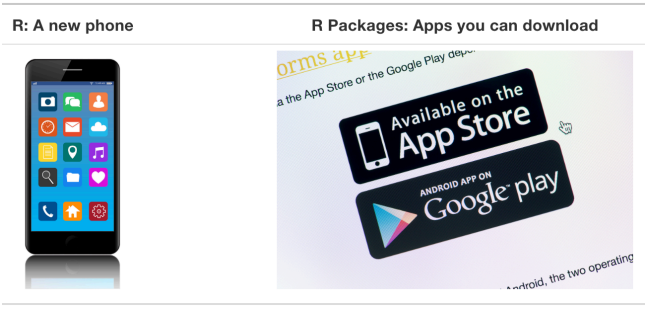
If you want to remove all objects from your current environment, you can run:

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

6.3 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.

6.3.1 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).

6.3.2 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).

6.3.3 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`

6.4 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_`.

6.4.1 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 2020"
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.

6.4.2 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.

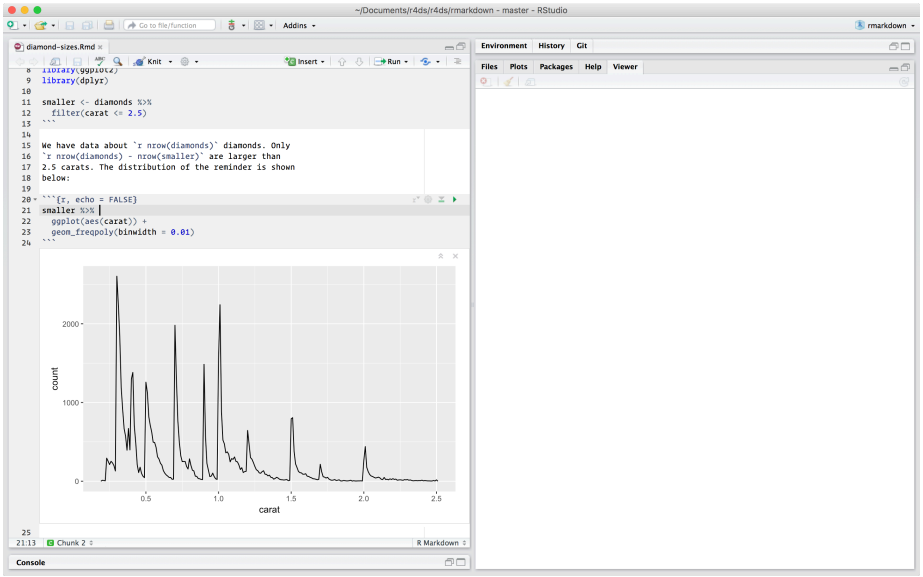
6.4.3 Code Chunks

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:



6.4.4 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.

6.4.5 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.

6.4.6 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.

6.4.7 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 7

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

7.1 Variables

7.1.1 Arithmetic operators

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

Operator	Description
+	addition
-	subtraction
	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (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.

7.1.2 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
```

7.1.3 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
```

7.1.4 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
```

7.2 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
```

7.2.1 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?

7.2.2 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
```

7.2.3 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
```

7.2.4 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.

7.2.5 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.

7.3 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

7.3.1 What Is that Type?

R is dynamically typed, meaning that it "guesses" what class a value is. Every piece of information in R has a type!

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
```

7.3.2 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?

7.3.3 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
```

7.3.4 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: Fixing the Code

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

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

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

7.4 Boolean Expressions

Boolean expressions are logical statements that are either true or false. For our purposes, we will often use Boolean expressions to compare quantities. For instance, the Boolean expression `1 < 2` is true, whereas the Boolean expression `1 > 2` is false.

When you type a Boolean expression in R, R will output `TRUE` if the expression is true and `FALSE` if the expression is false.

7.4.1 Boolean Operators

Operator	Description
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to
`%in%`	is an item of a set

Note that we use a double equal sign `==` to check whether two values are equal. Typing `a = b` would set the value of `a` equal to the value of `b`.

Here are some examples of Boolean expressions in action:

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

```
value_1 <- 1
value_1 > 0
#> [1] TRUE

value_2 <- value_1 + 10
value_1 + value_2 <= 12
#> [1] TRUE
```

7.4.2 Logical Operators

In practice, we often use two or more conditions to decide what to do. To combine different conditions, there are three logical operators:

Operator	Description
`x & y`	`x` AND `y`
`x | y`	`x` OR `y`
`!x`	Not `x`

First, `&` is similar to **AND** in English, that is, `x & y` is true only if both `x` and `y` are true. Second, is similar to **OR** in English, that is, A || B is false only if both `x` and `y` are false. Third, `!` is similar to **NOT** in English, that is, `!x` is true only if `x` is false.

```
x <- 10
y <- 20

# check to see if values are between 5 and 15
x > 5 & x < 15
#> [1] TRUE
y > 5 & y < 15
#> [1] FALSE

# more complex chains
(x > 5 & x < 15) & (x > y & y < 15)
#> [1] FALSE
(x > 5 & x < 15) | (x > y & y < 15)
#> [1] TRUE
(x > 5 & x < 15) & !(x > y & y < 15)
#> [1] TRUE
```

Different operator take different precendence. It is always a good practice to use brackets to control operation ordering.

7.4.3 Boolean Vectors

The nice thing about R is that you can use these comparison operators also on vectors. As with many expressions in R, the Boolean expressions discussed above are all *vectorized*. We will learn more about vectors and vectorization later in this class, but here is a quick 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`.

7.4.4 Boolean Vectors in Action

Boolean vectors are partially what makes R so magical. Check out the example below and examine each line. We will cover subsetting operations later, but pay attention to the work of boolean expressions and logical operators.

```
# An example
x <- c(1:10)
x[(x>8) | (x<5)]
#> [1]  1  2  3  4  9 10

# How it works
x <- c(1:10)
x
#>  [1]  1  2  3  4  5  6  7  8  9 10
x > 8
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
x < 5
#>  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
x > 8 | x < 5
#>  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
x[c(T,T,T,T,F,F,F,F,T,T)]
#> [1]  1  2  3  4  9 10
```

7.4.5 Challenges

Challenge 1.

In the partially written code below, `vector_1` and `vector_2` each contain 10 values. Using the fact that Boolean expressions are vectorized, write code that outputs:

1. A vector of length 10 such that element `i` of this vector equals `TRUE` if `vector_1[i]` is less than `vector_2[i]` and equals `FALSE` otherwise.
2. The number of times that element `i` of `vector_1` is less than element `i` of `vector_2`, using the `sum` function.

```
vector_1 <- c(1, 2, 4, 5, 3, 7, 8, 7, 1, 2)
vector_2 <- c(1, 3, 4, 4, 5, 10, 6, 8, 9, 1)

# YOUR CODE HERE
```

Chapter 8

Working with Data

This unit will cover the basics of working with data, including project workflow, data terms and concepts, importing data, and exploring data.

8.1 Project Workflow

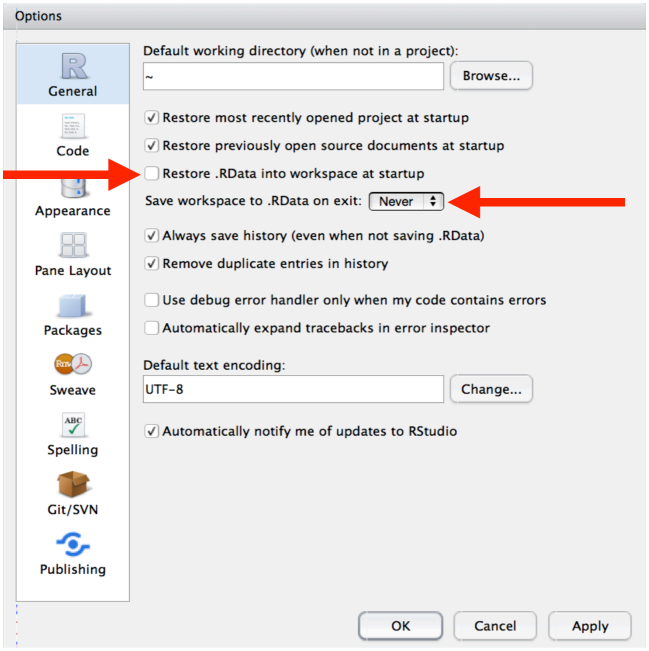
One day…

- 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 will 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.

8.1.1 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 will force you to capture all important interactions in your scripts.

8.1.2 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/materials"
```

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 they 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 is just a separator.

Mac/Linux vs. Windows

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

R can work with either type, no matter what platform you are 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 will see a `~` character in a path.

- In Mac/Linux, the `~` is a convenient shortcut to your **home directory** (`/users/rterman`).
- Windows does not 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`.

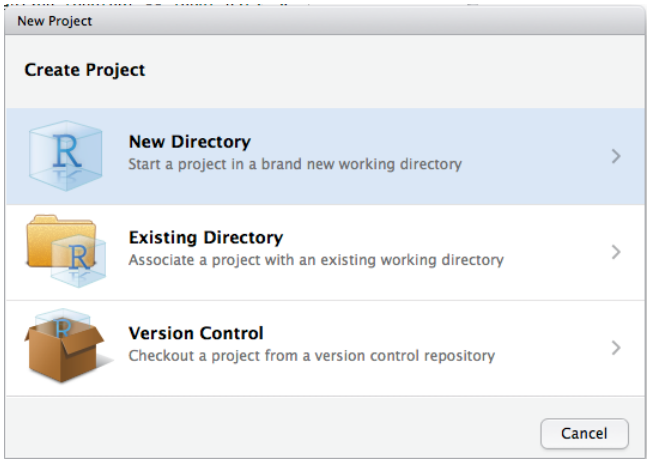
8.1.3 R Projects

As a beginning R user, it is 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 do not store it somewhere sensible, it will be hard to find in the future!

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

```
getwd()
#> [1] "/Users/rochelleterman/Desktop/materials"
```

Now whenever you refer to a file with a relative path, R will look for the file 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 is the same working directory and command history, and all the files you were working on are still open. Because you followed my instructions above, however, you will have a completely fresh environment, guaranteeing that you are starting with a clean slate.

8.1.4 File Organization

You should be saving all your files associated with your project in one directory. Here is 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. Do not ever change or overwrite the raw data!
- Export cleaned and altered data into a separate subdirectory.
- 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, which you can save in the `Results` subdirectory.

8.2 Importing and Exporting

8.2.1 Where Is 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 will discuss how to obtain data from web APIs and websites. For now, the rest of the unit discusses data that is stored **locally**.

8.2.2 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 dataset are indicated with `NA` or blank fields (do not use 99 or 77).

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

Second, you will need to know where the data file is relative to your working directory. If it is 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 dataframe 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 is 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 do not 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 customizability
mydataset <- read.table(file=, header=, sep=, quote=, dec=, fill=, stringsAsFactors=)
```

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

Reading Excel Files

Do not 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 is 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).

8.2.4 Exporting Data

You should never go from raw data to results in one script. Typically, you will 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 is 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 is 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..

Chapter 9

Data Transformation

9.1 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 will 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** are a set of values, each associated with a variable and an observation. Tabular data have rows (observations) and columns (variables). Tabular data are also called **rectangular** data or **spreadsheets**.

9.1.1 The Gapminder Dataset

This lesson discusses how to perform basic exploratory data analysis.

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

```
require(gapminder)
#> Loading required package: gapminder
gap <- gapminder
```

9.1.2 Structure and Dimensions

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

```
str(gap)
#> tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
#>  $ 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 [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
#>  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
#>  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 1288
#>  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

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

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

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.
```

Here are some more common ways to query info from a dataframe:

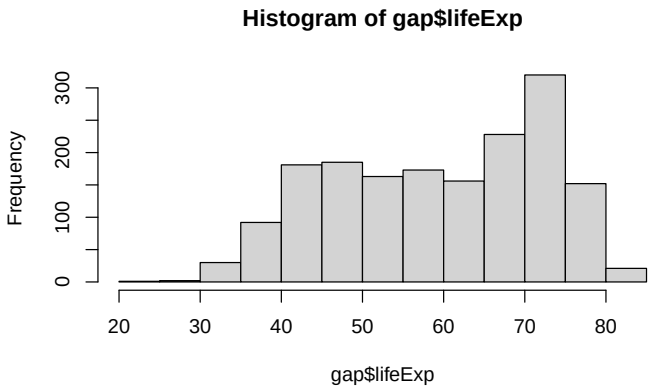
```
# Get number of rows and columns:
dim(gap)
#> [1] 1704    6

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

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

9.1.3 Variables

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

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



Data frames – unlike matrices in R – can hold variables of different flavors, such as character data (subject ID or name), quantitative data (white blood cell count), and categorical information (treated vs. untreated).

For example, the `year` variables is numeric, while the variables for `country` and `continent` hold categorical information, which is stored as a factor in R.

```
summary(gap$year)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1952    1966    1980    1980    1993    2007
summary(gap$country)
#>              Afghanistan                 Albania                 Algeria
#>                       12                      12                      12
#>                   Angola               Argentina               Australia
#>                       12                      12                      12
#>                  Austria                 Bahrain              Bangladesh
#>                       12                      12                      12
#>                  Belgium                   Benin                 Bolivia
#>                       12                      12                      12
#>   Bosnia and Herzegovina                Botswana                  Brazil
#>                       12                      12                      12
#>                 Bulgaria            Burkina Faso                 Burundi
#>                       12                      12                      12
#>                 Cambodia                Cameroon                  Canada
#>                       12                      12                      12
#> Central African Republic                    Chad                   Chile
#>                       12                      12                      12
#>                    China                Colombia                 Comoros
#>                       12                      12                      12
#>         Congo, Dem. Rep.             Congo, Rep.              Costa Rica
#>                       12                      12                      12
#>            Cote d'Ivoire                 Croatia                    Cuba
#>                       12                      12                      12
#>           Czech Republic                 Denmark                Djibouti
#>                       12                      12                      12
#>       Dominican Republic                 Ecuador                   Egypt
#>                       12                      12                      12
#>              El Salvador       Equatorial Guinea                 Eritrea
#>                       12                      12                      12
#>                 Ethiopia                 Finland                  France
#>                       12                      12                      12
#>                    Gabon                  Gambia                 Germany
#>                       12                      12                      12
#>                    Ghana                  Greece               Guatemala
#>                       12                      12                      12
#>                   Guinea           Guinea-Bissau                   Haiti
#>                       12                      12                      12
#>                 Honduras        Hong Kong, China                 Hungary
#>                       12                      12                      12
#>                  Iceland                   India               Indonesia
```

```
#>                       12                       12                       12
#>                     Iran                     Iraq                  Ireland
#>                       12                       12                       12
#>                   Israel                    Italy                  Jamaica
#>                       12                       12                       12
#>                    Japan                   Jordan                    Kenya
#>                       12                       12                       12
#>        Korea, Dem. Rep.             Korea, Rep.                   Kuwait
#>                       12                       12                       12
#>                  Lebanon                  Lesotho                  Liberia
#>                       12                       12                       12
#>                    Libya                Madagascar                   Malawi
#>                       12                       12                       12
#>                 Malaysia                     Mali                Mauritania
#>                       12                       12                       12
#>                Mauritius                   Mexico                 Mongolia
#>                       12                       12                       12
#>               Montenegro                  Morocco               Mozambique
#>                       12                       12                       12
#>                  Myanmar                  Namibia                    Nepal
#>                       12                       12                       12
#>              Netherlands              New Zealand                Nicaragua
#>                       12                       12                       12
#>                    Niger                  Nigeria                   Norway
#>                       12                       12                       12
#>                     Oman                 Pakistan                   Panama
#>                       12                       12                       12
#>                  (Other)
#>                      516
summary(gap$contintent)
#> Warning: Unknown or uninitialised column: `contintent`.
#> Length  Class   Mode
#>      0   NULL   NULL
```

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
```

9.1.4 Challenges

Challenge 1.

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

Challenge 2.

Report the number and name 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?

9.2 Introduction to dplyr

9.2.1 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 are going to focus on how to use the `dplyr` package for data transformation tasks.

For this unit, we will be working with the Gapminder dataset again.

```
library(tidyverse)
library(gapminder)

gap <- gapminder
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.
```

9.2.2 Why dplyr?

If you have ever used base R before, you know the following will calculate 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 is not 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 are going to cover 7 of the most commonly used `dplyr` functions. We will also cover pipes (`%>%`), which are used to combine those functions.

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

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

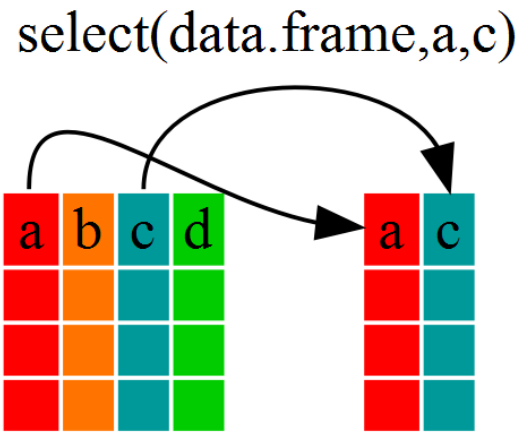
```
# not run
# install.packages('tidyverse')
require(tidyverse)
```

9.2.3 Select Columns with `select`

Imagine that we have 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)
head(year_country_gdp)
#> # A tibble: 6 x 3
#>    year country     gdpPercap
#>   <int> <fct>           <dbl>
#> 1  1952 Afghanistan      779.
#> 2  1957 Afghanistan      821.
#> 3  1962 Afghanistan      853.
#> 4  1967 Afghanistan      836.
```

```
#> 5  1972 Afghanistan      740.
#> 6  1977 Afghanistan      786.
```



If we open up `year_country_gdp`, we will 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")]
head(year_country_gdp)
#> # A tibble: 6 x 3
#>    year country     gdpPercap
#>   <int> <fct>           <dbl>
#> 1  1952 Afghanistan      779.
#> 2  1957 Afghanistan      821.
#> 3  1962 Afghanistan      853.
#> 4  1967 Afghanistan      836.
#> 5  1972 Afghanistan      740.
#> 6  1977 Afghanistan      786.
```

9.2.4 The Pipe

Above, we used what is called 'normal' grammar, but the strengths of `dplyr` lie in combining several functions 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 are not very

useful to us. Plus, these variables are often 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()
```

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

Let's repeat what we did above with the Gapminder dataset using pipes:

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

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 do not specify which data object we use in the call to `select()` since we have piped it in from the previous line.

Tips for Piping

1. Remember that you do not assign anything within the pipes — that is, you should not use `<-` inside the piped operation. Only use `<-` 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 is not indented, it probably has not 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`.

9.2.5 Filter Rows with `filter`

Now let's say we are 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.

NB: 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.

9.3 More `dplyr` functions

Where were we?

In the previous lesson, we used two very important verbs and an operator:

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

We also discussed `dplyr`'s role inside the tidyverse:

- `dplyr` is a core package in the tidyverse meta-package.
- Since we often make incidental usage of the others, we will load `dplyr` and the others via `library(tidyverse)`.

```
library(tidyverse)
library(gapminder)

gap <- gapminder
```

9.3.1 Use `mutate()` to Add New Variables

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

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

```
gap %>%
  mutate(gdp = pop * gdpPercap) %>%
  head()
#> # A tibble: 6 x 7
#>   country     continent  year lifeExp      pop gdpPercap          gdp
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.  6567086330.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.  7585448670.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.  8758855797.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.  9648014150.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.  9678553274.
#> 6 Afghanistan Asia       1977    38.4 14880372      786. 11697659231.
```

We can add multiple columns in one call:

```
gap %>%
  mutate(gdp = pop * gdpPercap,
         log_gdp = log(gdp)) %>%
  head()
#> # A tibble: 6 x 8
#>   country     continent  year lifeExp      pop gdpPercap          gdp log_gdp
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>   <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.  6567086330.    22.6
#> 2 Afghanistan Asia       1957    30.3  9240934      821.  7585448670.    22.7
#> 3 Afghanistan Asia       1962    32.0 10267083      853.  8758855797.    22.9
#> 4 Afghanistan Asia       1967    34.0 11537966      836.  9648014150.    23.0
#> 5 Afghanistan Asia       1972    36.1 13079460      740.  9678553274.    23.0
#> 6 Afghanistan Asia       1977    38.4 14880372      786. 11697659231.    23.2
```

9.3.2 Use `arrange()` to Row-order Data in a Principled Way

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

```
gap %>%
  arrange(year, country)
```

```
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Albania     Europe     1952    55.2  1282697     1601.
#> 3 Algeria     Africa     1952    43.1  9279525     2449.
#> 4 Angola      Africa     1952    30.0  4232095     3521.
#> 5 Argentina   Americas   1952    62.5 17876956     5911.
#> 6 Australia   Oceania    1952    69.1  8691212    10040.
#> # ... with 1,698 more rows
```

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

```
gap %>%
  filter(year == 2007) %>%
  arrange(lifeExp)
#> # A tibble: 142 x 6
#>    country      continent  year lifeExp      pop gdpPercap
#>    <fct>        <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Swaziland    Africa     2007    39.6  1133066     4513.
#> 2 Mozambique   Africa     2007    42.1 19951656      824.
#> 3 Zambia       Africa     2007    42.4 11746035     1271.
#> 4 Sierra Leone Africa     2007    42.6  6144562      863.
#> 5 Lesotho      Africa     2007    42.6  2012649     1569.
#> 6 Angola       Africa     2007    42.7 12420476     4797.
#> # ... with 136 more rows
```

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

```
gap %>%
  filter(year == 2007) %>%
  arrange(desc(lifeExp))
#> # A tibble: 142 x 6
#>    country          continent  year lifeExp       pop gdpPercap
#>    <fct>            <fct>     <int>   <dbl>     <int>     <dbl>
#> 1 Japan            Asia       2007    82.6 127467972    31656.
#> 2 Hong Kong, China Asia       2007    82.2   6980412    39725.
#> 3 Iceland          Europe     2007    81.8    301931    36181.
#> 4 Switzerland      Europe     2007    81.7   7554661    37506.
#> 5 Australia        Oceania    2007    81.2  20434176    34435.
#> 6 Spain            Europe     2007    80.9  40448191    28821.
#> # ... with 136 more rows
```

I advise that your analyses NEVER rely on rows or variables being in a specific order. But it is still true that human beings write the code, and the interactive development process can be much nicer if you reorder the rows of your data

as you go along. Also, once you are preparing tables for human eyeballs, it is imperative that you step up and take control of row order.

9.3.3 Use `rename()` to Rename Variables

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

```
gap %>%
  rename(life_exp = lifeExp,
         gdp_percap = gdpPercap)
#> # A tibble: 1,704 x 6
#>    country     continent  year life_exp      pop gdp_percap
#>    <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.
#> # ... with 1,698 more rows
```

9.3.4 Use `select()` to Rename and Reposition Variables

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

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

```
gap %>%
  filter(country == "Burundi", year > 1996) %>%
  select(yr = year, lifeExp, gdpPercap) %>%
  select(gdpPercap, everything())
#> # A tibble: 3 x 3
#>   gdpPercap    yr lifeExp
#>       <dbl> <int>   <dbl>
#> 1      463.  1997    45.3
#> 2      446.  2002    47.4
#> 3      430.  2007    49.6
```

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

9.3.5 Use `count()` to Count Variable Quantities

Finally, let's say we want to examine if the number of countries covered in the Gapminder dataset 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 dataset:

```
gap %>%
  dplyr::count(year)
#> # A tibble: 12 x 2
#>    year     n
#>   <int> <int>
#> 1  1952   142
#> 2  1957   142
#> 3  1962   142
#> 4  1967   142
#> 5  1972   142
#> 6  1977   142
#> # ... with 6 more rows
```

We can confirm that each year in the dataset 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 %>%
  filter(gdpPercap >= 10000) %>%
  dplyr::count(year)
#> # A tibble: 12 x 2
#>    year     n
#>   <int> <int>
#> 1  1952     7
#> 2  1957    12
#> 3  1962    19
#> 4  1967    22
#> 5  1972    32
#> 6  1977    41
#> # ... with 6 more rows
```

```
library(tidyverse)
library(gapminder)

gap <- gapminder
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.
```

9.4 Calculating across Groups

A common task you will 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
```

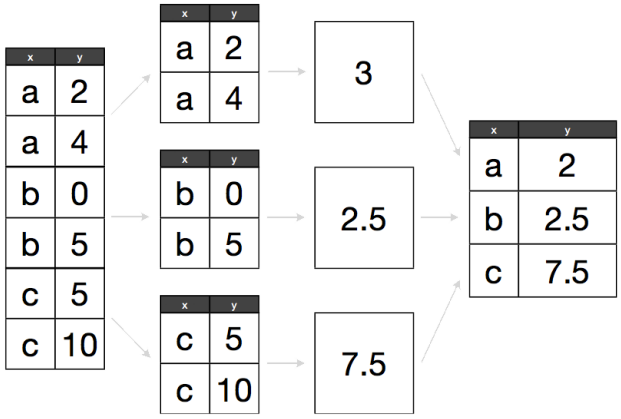
That is a lot of repetition! To make matters worse, what if we wanted to add these values to our original data frame as a new column? We would have to write something like this:

```
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 are 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.

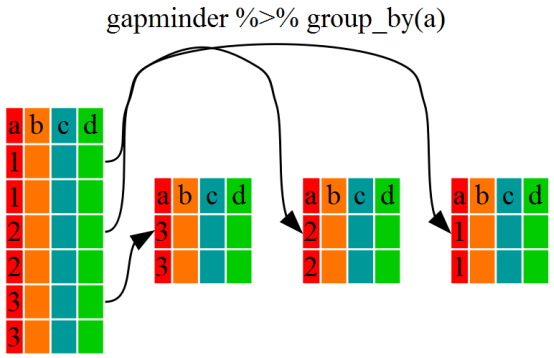
First, let's remove the column we just made:

```
gap <- gapminder
```

9.4.1 Use `group_by` to Create a Grouped Data

We have 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 criterium 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).



9.4.2 Summarize Across Groups with `summarize`

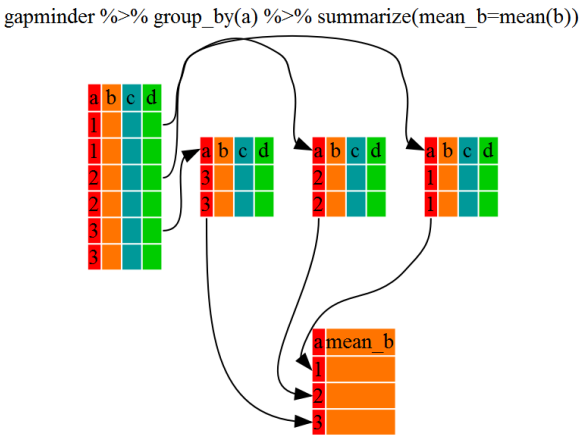
`group_by()` on its own is not particularly interesting. It is much more exciting used in conjunction with the `summarize()` function.

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.

```
gap %>%
  group_by(continent) %>%
  summarize(mean_gdpPercap = mean(gdpPercap))
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 5 x 2
#>   continent mean_gdpPercap
#>   <fct>              <dbl>
#> 1 Africa             2194.
#> 2 Americas           7136.
#> 3 Asia               7902.
#> 4 Europe            14469.
#> 5 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`:

```
gap %>%
  group_by(continent, year) %>%
  summarize(mean_gdpPercap = mean(gdpPercap)) %>%
  head()
#> `summarise()` regrouping output by 'continent' (override with `.groups` argument)
#> # A tibble: 6 x 3
#> # Groups:   continent [1]
#>   continent  year mean_gdpPercap
#>   <fct>     <int>          <dbl>
#> 1 Africa     1952          1253.
#> 2 Africa     1957          1385.
#> 3 Africa     1962          1598.
#> 4 Africa     1967          2050.
#> 5 Africa     1972          2340.
#> 6 Africa     1977          2586.
```

That is already quite powerful, but it gets even better! You are not limited to defining only one new variable in `summarize()`.

```
gap %>%
  group_by(continent, year) %>%
  summarize(mean_gdpPercap = mean(gdpPercap),
            sd_gdpPercap = sd(gdpPercap),
            mean_pop = mean(pop),
            sd_pop = sd(pop))
#> `summarise()` regrouping output by 'continent' (override with `.groups` argument)
```

```
#> # A tibble: 60 x 6
#> # Groups:   continent [5]
#>   continent  year mean_gdpPercap sd_gdpPercap mean_pop    sd_pop
#>   <fct>     <int>          <dbl>        <dbl>    <dbl>     <dbl>
#> 1 Africa     1952          1253.         983. 4570010.  6317450.
#> 2 Africa     1957          1385.        1135. 5093033.  7076042.
#> 3 Africa     1962          1598.        1462. 5702247.  7957545.
#> 4 Africa     1967          2050.        2848. 6447875.  8985505.
#> 5 Africa     1972          2340.        3287. 7305376. 10130833.
#> 6 Africa     1977          2586.        4142. 8328097. 11585184.
#> # ... with 54 more rows
```

9.4.3 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 data frame that you pass into it.

```
gap %>%
  group_by(continent, year) %>%
  mutate(mean_gdpPercap = mean(gdpPercap),
            sd_gdpPercap = sd(gdpPercap),
            mean_pop = mean(pop),
            sd_pop = sd(pop))
#> # A tibble: 1,704 x 10
#> # Groups:   continent, year [60]
#>   country continent  year lifeExp    pop gdpPercap mean_gdpPercap sd_gdpPercap
#>   <fct>   <fct>     <int>   <dbl>  <int>     <dbl>          <dbl>        <dbl>
#> 1 Afghan~ Asia       1952    28.8 8.43e6      779.          5195.       18635.
#> 2 Afghan~ Asia       1957    30.3 9.24e6      821.          5788.       19507.
#> 3 Afghan~ Asia       1962    32.0 1.03e7      853.          5729.       16416.
#> 4 Afghan~ Asia       1967    34.0 1.15e7      836.          5971.       14063.
#> 5 Afghan~ Asia       1972    36.1 1.31e7      740.          8187.       19088.
#> 6 Afghan~ Asia       1977    38.4 1.49e7      786.          7791.       11816.
#> # ... with 1,698 more rows, and 2 more variables: mean_pop <dbl>, sd_pop <dbl>
```

We can also use `mutate()` to create new variables prior to (or even after) summarizing the information.

```
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))
#> `summarise()` regrouping output by 'continent' (override with `.groups` argument)
#> # A tibble: 60 x 8
#> # Groups:   continent [5]
#>   continent  year mean_gdpPercap sd_gdpPercap mean_pop sd_pop mean_gdp_billion
#>   <fct>     <int>          <dbl>        <dbl>    <dbl>  <dbl>            <dbl>
#> 1 Africa     1952          1253.         983. 4570010. 6.32e6             5.99
#> 2 Africa     1957          1385.        1135. 5093033. 7.08e6             7.36
#> 3 Africa     1962          1598.        1462. 5702247. 7.96e6             8.78
#> 4 Africa     1967          2050.        2848. 6447875. 8.99e6            11.4
#> 5 Africa     1972          2340.        3287. 7305376. 1.01e7            15.1
#> 6 Africa     1977          2586.        4142. 8328097. 1.16e7            18.7
#> # ... with 54 more rows, and 1 more variable: sd_gdp_billion <dbl>
```

`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 will 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 will simply create an output data frame with one row (i.e., the whole input data frame is a single group).

9.5 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, and (b) arrange the data frame by the column you just created in descending order (so that the relatively richest country-years are listed first).

Acknowledgments

Some of the materials in this module were adapted from:

- Software Carpentry.
- R bootcamp at UC Berkeley.

Chapter 10

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.

10.1 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)
)
wide
#>      name time1 time2 time3
#> 1  Wilbur    67    56    70
#> 2 Petunia    80    90    67
#> 3 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)
)
long
#>      name time heartrate
#> 1  Wilbur    1        67
#> 2 Petunia    1        80
#> 3 Gregory    1        64
#> 4  Wilbur    2        56
#> 5 Petunia    2        90
#> 6 Gregory    2        50
#> 7  Wilbur    3        70
#> 8 Petunia    3        67
#> 9 Gregory    3        10
```

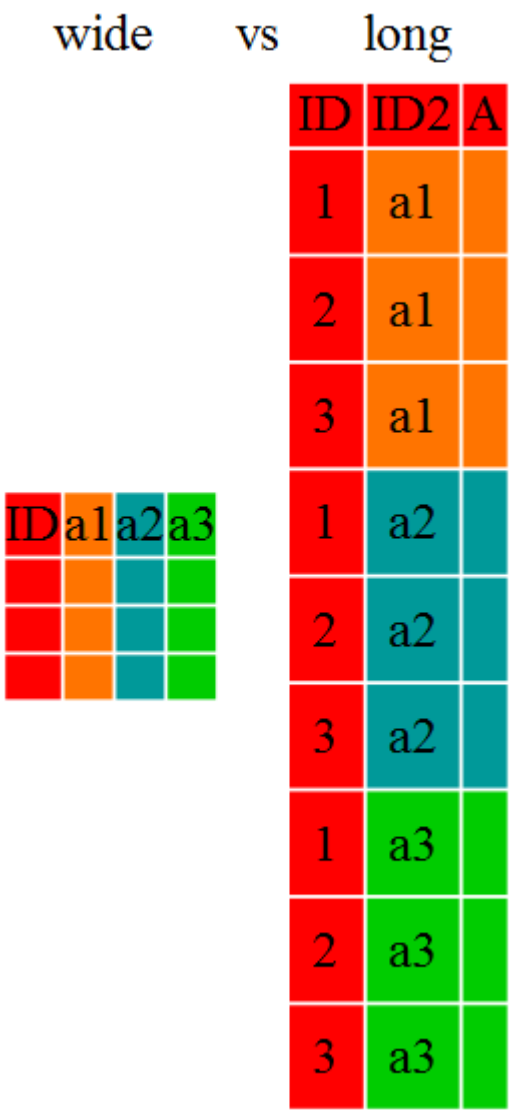
Question: Which one of these do you think is the *tidy* format?

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.

10.2 Tidying the Gapminder Data

Let's look at the structure of our original `gapminder` dataframe:

```
library(gapminder)

gap <- gapminder
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.
```

Question: Is this dataframe **wide** or **long**?

Answer: This dataframe 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.

10.3 `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)
```

10.3.1 `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)
head(gap_wide)
#>   continent       country gdpPercap_1952 gdpPercap_1957 gdpPercap_1962
#> 1    Africa       Algeria           2449           3014           2551
#> 2    Africa        Angola           3521           3828           4269
#> 3    Africa         Benin           1063            960            949
#> 4    Africa      Botswana            851            918            984
#> 5    Africa Burkina Faso            543            617            723
#> 6    Africa       Burundi            339            380            355
#>   gdpPercap_1967 gdpPercap_1972 gdpPercap_1977 gdpPercap_1982 gdpPercap_1987
#> 1           3247           4183           4910           5745           5681
#> 2           5523           5473           3009           2757           2430
#> 3           1036           1086           1029           1278           1226
#> 4           1215           2264           3215           4551           6206
#> 5            795            855            743            807            912
#> 6            413            464            556            560            622
#>   gdpPercap_1992 gdpPercap_1997 gdpPercap_2002 gdpPercap_2007 lifeExp_1952
#> 1           5023           4797           5288           6223         43.1
#> 2           2628           2277           2773           4797         30.0
#> 3           1191           1233           1373           1441         38.2
#> 4           7954           8647          11004          12570         47.6
#> 5            932            946           1038           1217         32.0
#> 6            632            463            446            430         39.0
#>   lifeExp_1957 lifeExp_1962 lifeExp_1967 lifeExp_1972 lifeExp_1977 lifeExp_1982
#> 1         45.7         48.3         51.4         54.5         58.0         61.4
#> 2         32.0         34.0         36.0         37.9         39.5         39.9
#> 3         40.4         42.6         44.9         47.0         49.2         50.9
#> 4         49.6         51.5         53.3         56.0         59.3         61.5
#> 5         34.9         37.8         40.7         43.6         46.1         48.1
#> 6         40.5         42.0         43.5         44.1         45.9         47.5
#>   lifeExp_1987 lifeExp_1992 lifeExp_1997 lifeExp_2002 lifeExp_2007 pop_1952
#> 1         65.8         67.7         69.2         71.0         72.3  9279525
#> 2         39.9         40.6         41.0         41.0         42.7  4232095
#> 3         52.3         53.9         54.8         54.4         56.7  1738315
#> 4         63.6         62.7         52.6         46.6         50.7   442308
#> 5         49.6         50.3         50.3         50.6         52.3  4469979
#> 6         48.2         44.7         45.3         47.4         49.6  2445618
#>   pop_1957 pop_1962 pop_1967 pop_1972 pop_1977 pop_1982 pop_1987 pop_1992
#> 1 10270856 11000948 12760499 14760787 17152804 20033753 23254956 26298373
#> 2  4561361  4826015  5247469  5894858  6162675  7016384  7874230  8735988
#> 3  1925173  2151895  2427334  2761407  3168267  3641603  4243788  4981671
#> 4   474639   512764   553541   619351   781472   970347  1151184  1342614
#> 5  4713416  4919632  5127935  5433886  5889574  6634596  7586551  8878303
```

```
#> 6  2667518  2961915  3330989  3529983  3834415  4580410  5126023  5809236
#>   pop_1997 pop_2002 pop_2007
#> 1 29072015 31287142 33333216
#> 2  9875024 10866106 12420476
#> 3  6066080  7026113  8078314
#> 4  1536536  1630347  1639131
#> 5 10352843 12251209 14326203
#> 6  6121610  7021078  8390505
```

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)
head(gap_long)
#>   continent      country   obstype_year obs_values
#> 1    Africa      Algeria gdpPercap_1952       2449
#> 2    Africa       Angola gdpPercap_1952       3521
#> 3    Africa        Benin gdpPercap_1952       1063
#> 4    Africa     Botswana gdpPercap_1952        851
#> 5    Africa Burkina Faso gdpPercap_1952        543
#> 6    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'))
head(gap_long)
#>   continent      country obstype_year obs_values
#> 1    Africa      Algeria     pop_1952    9279525
#> 2    Africa       Angola     pop_1952    4232095
#> 3    Africa        Benin     pop_1952    1738315
#> 4    Africa     Botswana     pop_1952     442308
#> 5    Africa Burkina Faso     pop_1952    4469979
#> 6    Africa      Burundi     pop_1952    2445618

# 2. With the - operator:
gap_long <- gap_wide %>%
  gather(obstype_year, obs_values, -continent, -country)
head(gap_long)
#>   continent      country   obstype_year obs_values
#> 1    Africa      Algeria gdpPercap_1952       2449
#> 2    Africa       Angola gdpPercap_1952       3521
#> 3    Africa        Benin gdpPercap_1952       1063
#> 4    Africa     Botswana gdpPercap_1952        851
#> 5    Africa Burkina Faso gdpPercap_1952        543
#> 6    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.

10.3.2 `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))
head(gap_long_sep)
#>   continent      country  obs_type year obs_values
```

```
#> 1    Africa      Algeria gdpPercap 1952       2449
#> 2    Africa       Angola gdpPercap 1952       3521
#> 3    Africa        Benin gdpPercap 1952       1063
#> 4    Africa     Botswana gdpPercap 1952        851
#> 5    Africa Burkina Faso gdpPercap 1952        543
#> 6    Africa      Burundi gdpPercap 1952        339
```

10.3.3 `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)
head(gap_medium)
#>   continent country year gdpPercap lifeExp      pop
#> 1    Africa Algeria 1952      2449    43.1  9279525
#> 2    Africa Algeria 1957      3014    45.7 10270856
#> 3    Africa Algeria 1962      2551    48.3 11000948
#> 4    Africa Algeria 1967      3247    51.4 12760499
#> 5    Africa Algeria 1972      4183    54.5 14760787
#> 6    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
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.
# Rearrange columns:
gap_medium <- gap_medium %>%
  select(country, continent, year, lifeExp, pop, gdpPercap)
head(gap_medium)
#>   country continent year lifeExp      pop gdpPercap
#> 1 Algeria    Africa 1952    43.1  9279525      2449
#> 2 Algeria    Africa 1957    45.7 10270856      3014
```

```
#> 3 Algeria    Africa 1962    48.3 11000948      2551
#> 4 Algeria    Africa 1967    51.4 12760499      3247
#> 5 Algeria    Africa 1972    54.5 14760787      4183
#> 6 Algeria    Africa 1977    58.0 17152804      4910
# Arrange by country, continent, and year:
gap_medium <- gap_medium %>%
  arrange(country,continent,year)
head(gap_medium)
#>       country continent year lifeExp      pop gdpPercap
#> 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
```

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.

10.4 Dealing with Missing Data

A common challenge in applying quantitative tools to social science problems is dealing with missing data. You'll see a variety of ways the creators of data sets designate that a piece of data is missing - for example, `NA`, `-99`, or `-77` are sometimes used to denote a missing piece of data. We recommend using `NA`, which has a variety of associated functions that are useful when transforming missing data.

10.4.1 `na_if`

You can use `na_if` to replace certain pieces of data with `NA`. Consider the case where `lifeExp` values below 35 are missing and simply filled in with "unknown":

```
gap_medium$lifeExp[gap_medium$lifeExp < 35] <- "unknown"
```

This is problematic for many reasons, including that we cannot perform simple mathematical functions on columns with both number and character values:

```
mean(gap_medium$lifeExp, na.rm = TRUE)
#> Warning in mean.default(gap_medium$lifeExp, na.rm = TRUE): argument is not
#> numeric or logical: returning NA
#> [1] NA
```

However, NAs are different in the sense that they can exist in a numeric vector, and therefore you still can perform math functions (R will omit those observations with NA). Below, we replace the "unknown" values with NA, and we can then calculate the mean lifeExp.

```
gap_medium <- gap_medium %>%
  mutate(lifeExp = na_if(lifeExp, "unknown"),
         lifeExp = as.double(lifeExp))

mean(gap_medium$lifeExp, na.rm = TRUE)
#> [1] 60
```

10.4.2 Replace NA values with replace_na

Sometimes, you will want to replace all NA values in your data (for instance, maybe you know that the true value of anything coded as NA is actually 30). replace_na is a simple command that will replace all NAs with a new value.

```
gap_na_replaced <- gap_medium %>%
  mutate(lifeExp = replace_na(lifeExp, 30))

head(gap_na_replaced)
#>       country continent year lifeExp      pop gdpPercap
#> 1 Afghanistan      Asia 1952    30.0  8425333       779
#> 2 Afghanistan      Asia 1957    30.0  9240934       821
#> 3 Afghanistan      Asia 1962    30.0 10267083       853
#> 4 Afghanistan      Asia 1967    30.0 11537966       836
#> 5 Afghanistan      Asia 1972    36.1 13079460       740
#> 6 Afghanistan      Asia 1977    38.4 14880372       786
```

10.5 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…

10.6 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.

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

Chapter 11

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.

11.1 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)
)

messy
#>   county state cnty_pop state_pop region
#> 1  36037    NY  3817735  43320903      1
#> 2  36038    NY   422999  43320903      1
```

```
#> 3  36039    NY   324920        NA       1
#> 4  36040  <NA>   143432  43320903       1
#> 5     NA  <NA>       NA  43320903       1
#> 6  37001    VA  3228290   7173000       3
#> 7  37002    VA   449499   7173000       3
#> 8  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
)
counties
#>   county state county_pop
#> 1  36037    NY    3817735
#> 2  36038    NY     422999
#> 3  36039    NY     324920
#> 4  36040    NY     143432
#> 5  37001    VA    3228290
#> 6  37002    VA     449499
#> 7  37003    VA     383888

states <- data.frame(
  state = c("NY", "VA"),
  state_pop = c(43320903, 7173000),
  region = c(1, 3), stringsAsFactors = F
)

states
#>   state state_pop region
#> 1    NY  43320903      1
#> 2    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.

11.2 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**.

11.3 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")

counties_states
#>   county state county_pop state_pop region
#> 1  36037    NY    3817735  43320903      1
#> 2  36038    NY     422999  43320903      1
#> 3  36039    NY     324920  43320903      1
#> 4  36040    NY     143432  43320903      1
#> 5  37001    VA    3228290   7173000      3
#> 6  37002    VA     449499   7173000      3
#> 7  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.

11.4 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 Polity IV dataset.

```
gap <- gapminder

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 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")

head(gap1)
#> # A tibble: 6 x 7
#>   country     continent  year lifeExp      pop gdpPercap polity2
#>   <chr>       <fct>     <int>   <dbl>    <int>     <dbl>   <int>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.     -10
#> 2 Afghanistan Asia       1957    30.3  9240934      821.     -10
#> 3 Afghanistan Asia       1962    32.0 10267083      853.     -10
#> 4 Afghanistan Asia       1967    34.0 11537966      836.      -7
#> 5 Afghanistan Asia       1972    36.1 13079460      740.      -7
#> 6 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)
head(ciri)
#>          CTRY YEAR PHYSINT
#> 1 Afghanistan 1981       0
#> 2 Afghanistan 1982       0
#> 3 Afghanistan 1983       0
#> 4 Afghanistan 1984       0
#> 5 Afghanistan 1985       0
#> 6 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"))

head(gap2)
#> # A tibble: 6 x 8
#>   country     continent  year lifeExp      pop gdpPercap polity2 PHYSINT
#>   <chr>       <fct>     <int>   <dbl>    <int>     <dbl>   <int>   <int>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.     -10      NA
#> 2 Afghanistan Asia       1957    30.3  9240934      821.     -10      NA
#> 3 Afghanistan Asia       1962    32.0 10267083      853.     -10      NA
#> 4 Afghanistan Asia       1967    34.0 11537966      836.      -7      NA
#> 5 Afghanistan Asia       1972    36.1 13079460      740.      -7      NA
#> 6 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:

```
tail(gap2)
#> # A tibble: 6 x 8
#>   country  continent  year lifeExp      pop gdpPercap polity2 PHYSINT
#>   <chr>    <fct>     <int>   <dbl>    <int>     <dbl>   <int>   <int>
#> 1 Zimbabwe Africa     1982    60.4  7636524      789.       4       5
#> 2 Zimbabwe Africa     1987    62.4  9216418      706.      -6       5
#> 3 Zimbabwe Africa     1992    60.4 10704340      693.      -6       5
#> 4 Zimbabwe Africa     1997    46.8 11404948      792.      -6       6
#> 5 Zimbabwe Africa     2002    40.0 11926563      672.      -4       2
#> 6 Zimbabwe Africa     2007    43.5 12311143      470.      -4       1
```

11.5 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.

11.6 Challenges

Challenge 1.

Merge the Polity IV and CIRI datasets, keeping all observations in Polity IV. 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 12

Plotting

> "Make it informative, then make it pretty"

There are two major sets of tools for creating plots in R:

- 1. Base, which comes 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.

12.1 The Dataset

For the following examples, we will be using the `gapminder` dataset we have used previously. Gapminder is a country-year dataset with information on life expectancy, among other things.

```
library(gapminder)
gap <- gapminder
```

12.2 R Base Graphics

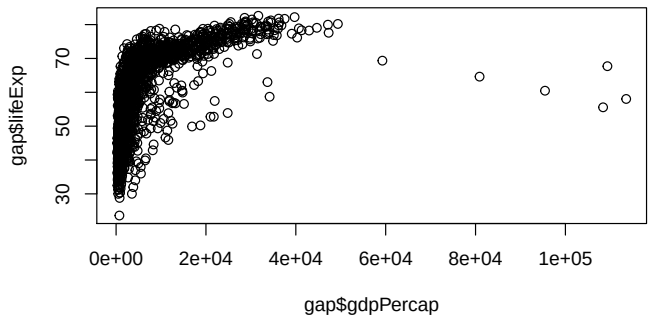
The **basic** call takes the following form:

```
plot(x=, y=)
```

We will also introduce a base R command to help us with creating these plots. To reference a specific column in a dataset, we use a `$` with the following syntax:

`dataset$column`. See this in action below, where we plot `gdpPercap` against `lifeExp`:

```
plot(x = gap$gdpPercap, y = gap$lifeExp)
```



12.2.1 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")
```

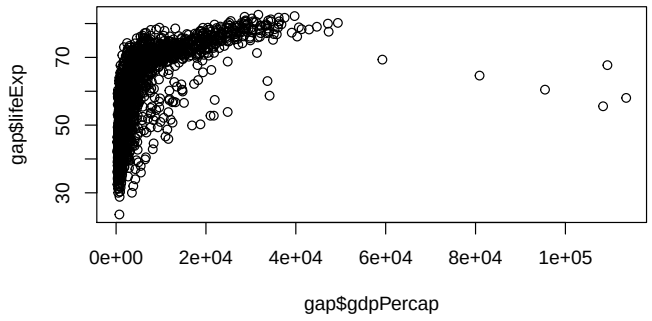


Figure 12.1:

- `"l"`: Line graphs.

```
# Note that "line" does not create a smooth 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")
```

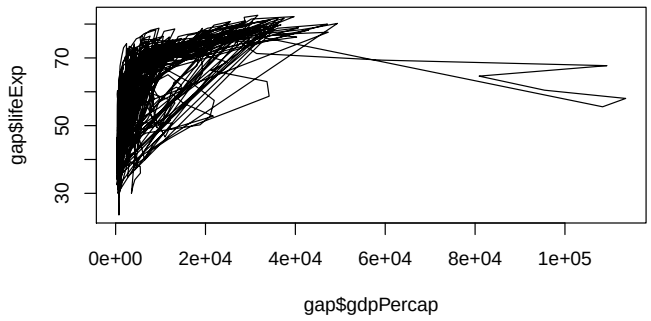


Figure 12.2:

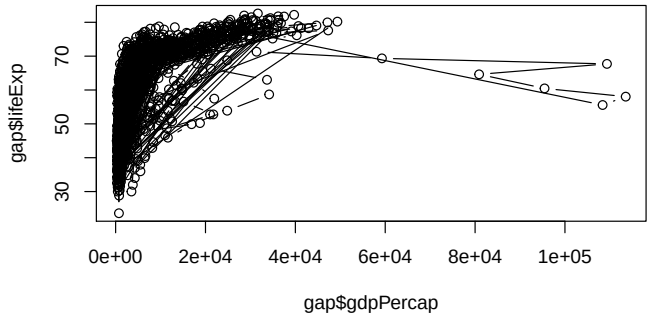
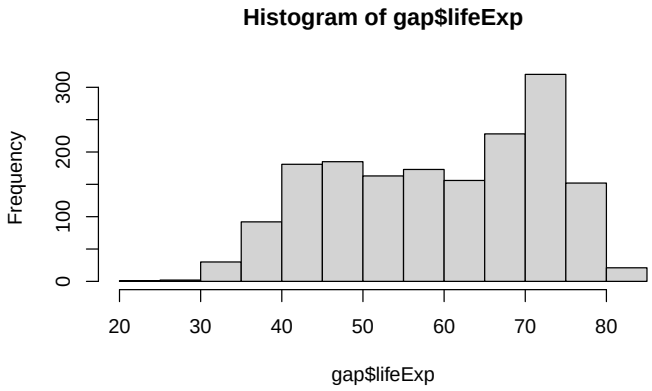


Figure 12.3:

12.2.2 Histograms and Density Plots

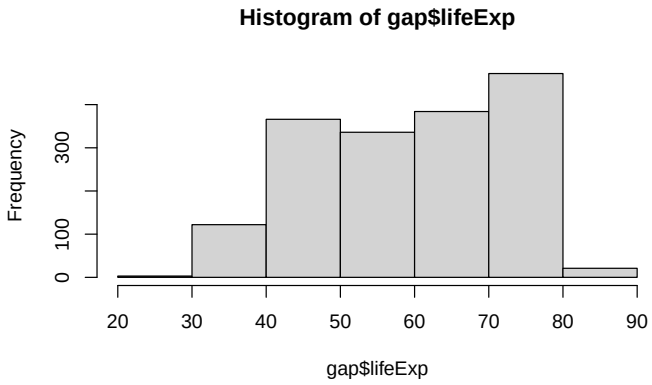
Histograms display the frequency of different values of a variable.

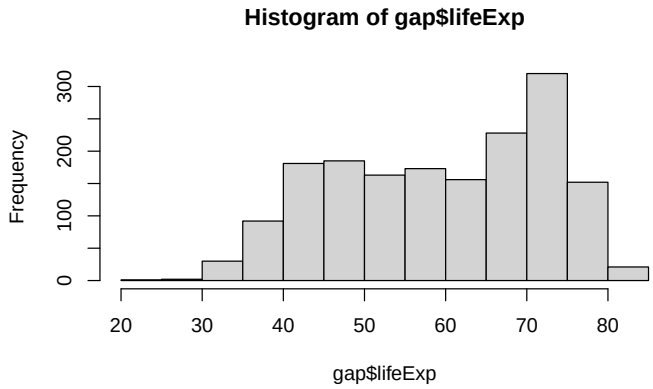
```
hist(x=gap$lifeExp)
```



Histograms require a `breaks` argument, which determines 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 continuous 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)
```

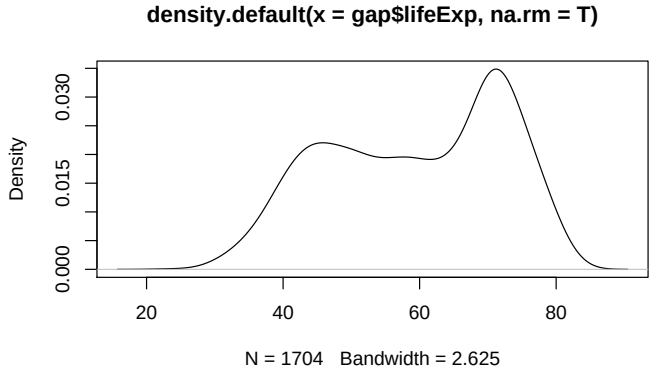
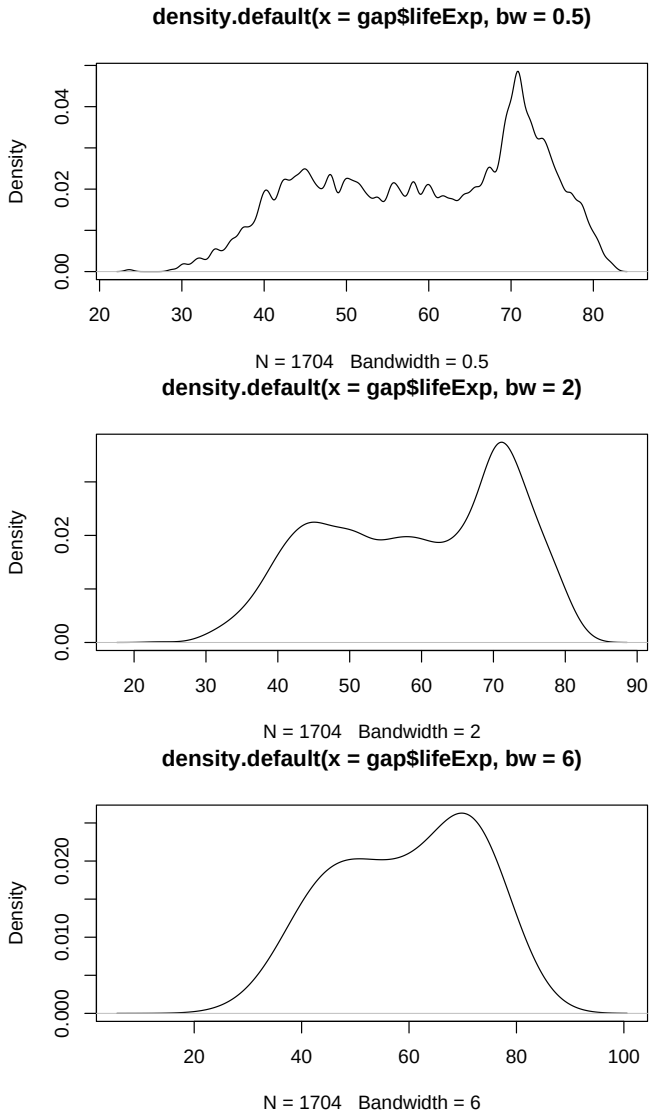


Figure 12.4:

Density passes a `bw` parameter, which determines 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))
```



12.2.3 Labels

Here is 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
```

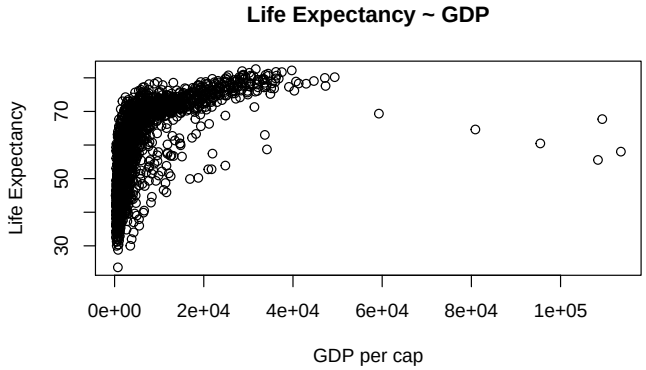


Figure 12.5:

12.2.4 Axis and Size Scaling

Currently it is 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 is the basic call with popular scaling arguments:

```
plot(x=, y=, type="", xlim=, ylim=, cex=)
```

From the previous example...

```
# 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)
```

12.2.5 Graphical Parameters

We can change the points with a number of graphical options:

```
plot(x=, y=, type="", col="", pch=, lty=, lwd=)
```

- Colors

```
library(dplyr)
colors() %>%
  head(20)# View first 20 elements of the color vector
```

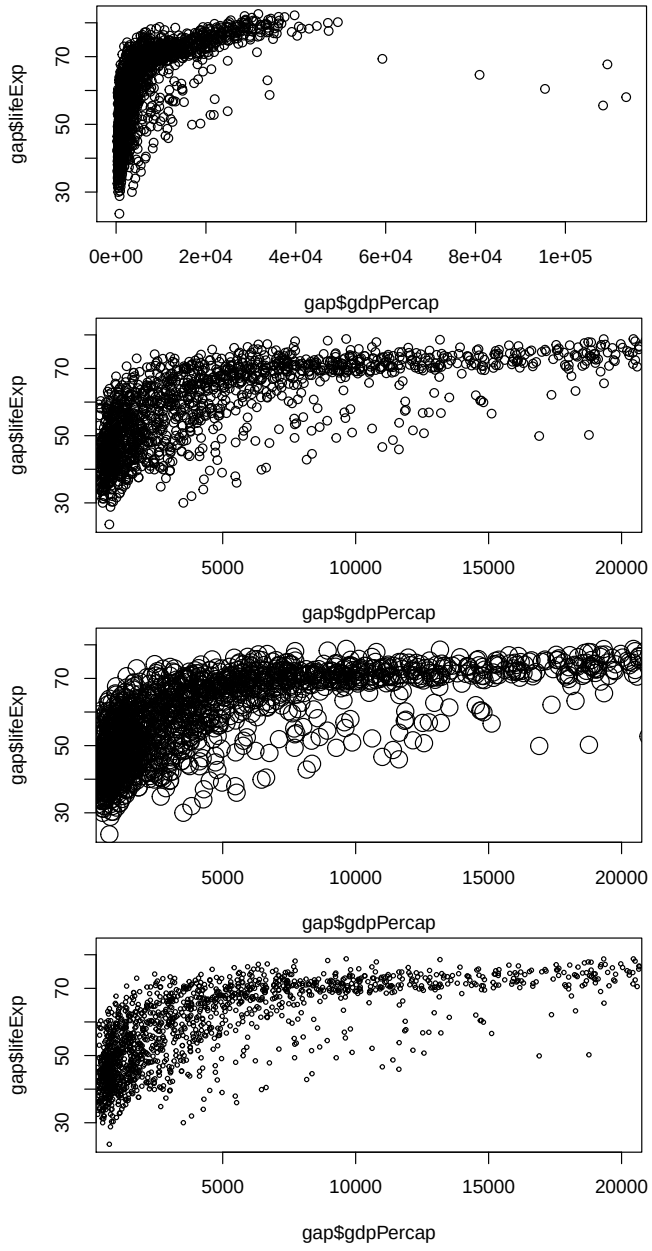

gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap
gap$lifeExp
30
50
70
5000
10000
15000
20000
gap$gdpPercap
gap$lifeExp
30
50
70
5000
10000
15000
20000
gap$gdpPercap
gap$lifeExp
30
50
70
5000
10000
15000
20000
gap$gdpPercap

Figure 12.6:

```
#>   [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"
```

Another option: R Color Infographic

```
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p", col="aquamarine1")
```

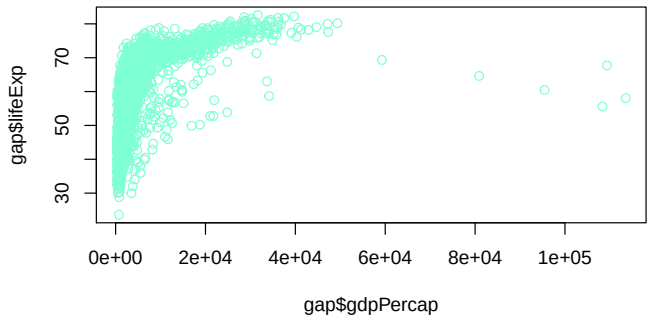


Figure 12.7:

- 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
plot(x = gap$gdpPercap, y = gap$lifeExp, type="l", lty=3, lwd=2)
```

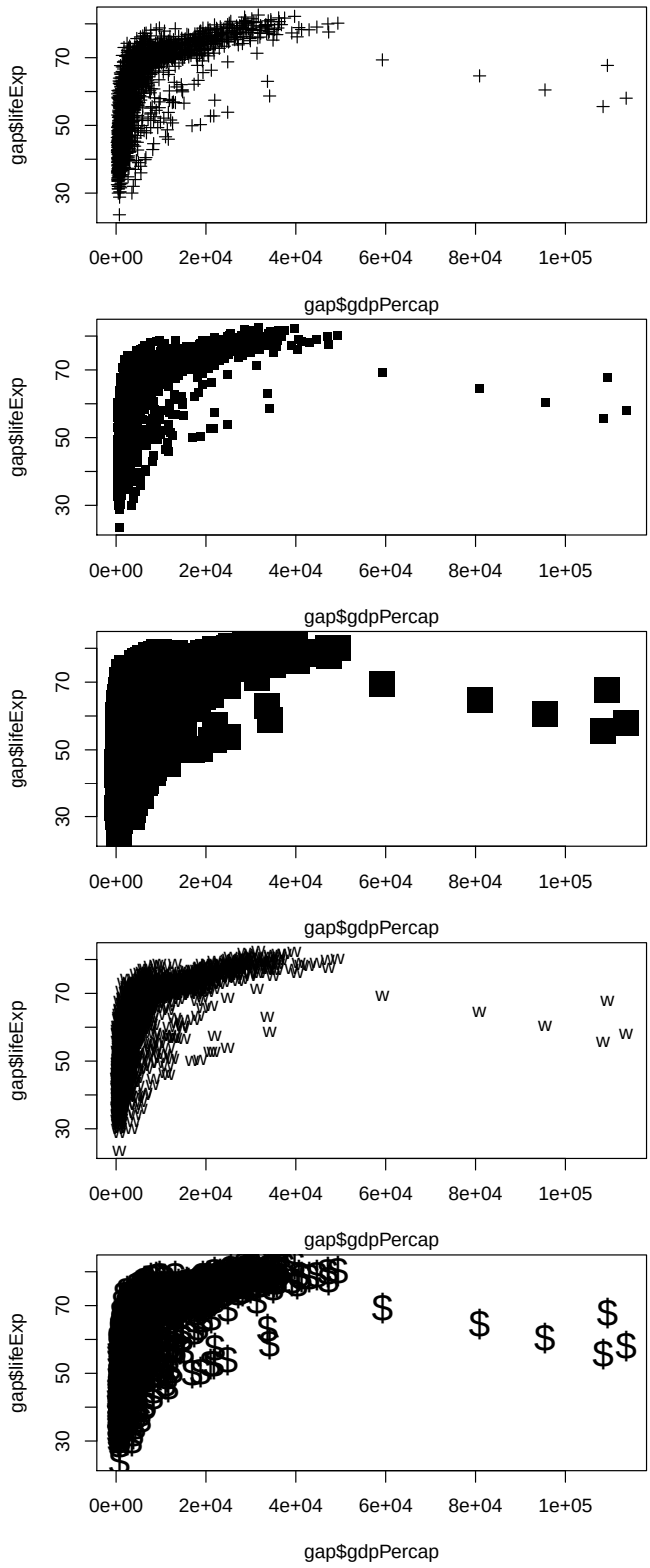

gap$lifeExp
gap$gdpPercap
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
30
50
70

Figure 12.8:

```
# 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)
```

12.2.6 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
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

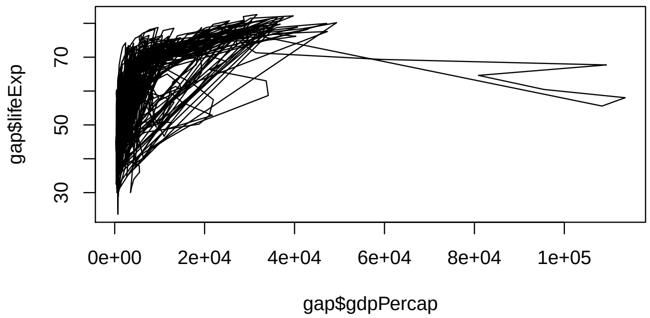
```
# Plot the line
plot(x = gap$gdpPercap, y = gap$lifeExp, type="p")
# Now the guides
abline(v=40000, h=75, lty=2)
```

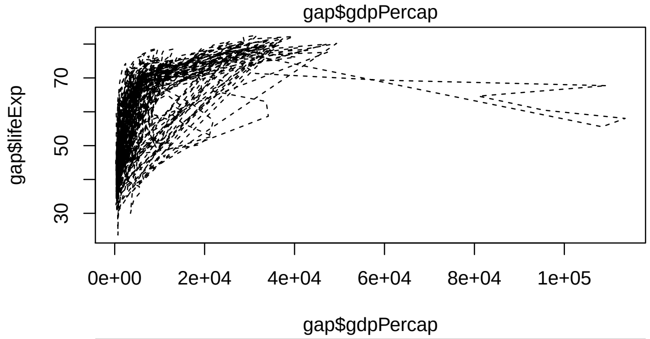
```
library(tidyverse)
library(gapminder)

gap <- gapminder
```

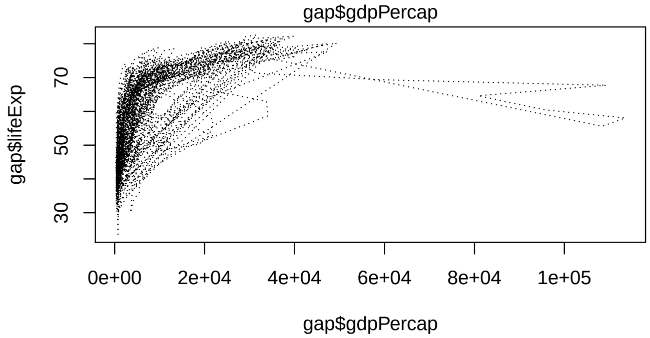
12.3 ggplot2

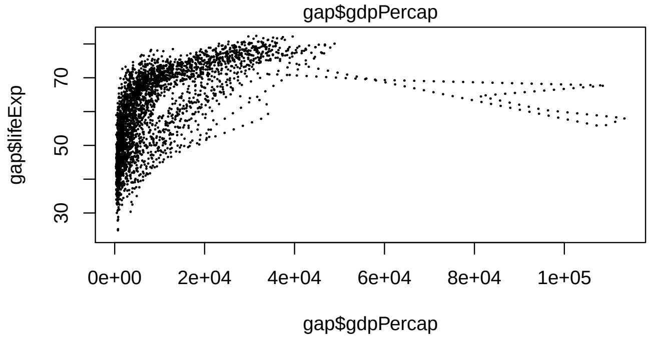
Why `ggplot`?

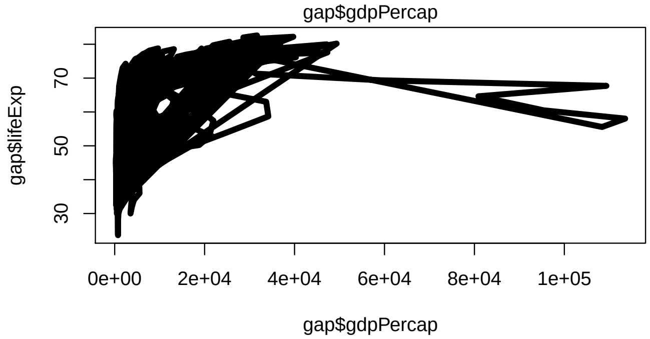
- More elegant and compact code than R base graphics.
- More aesthetically pleasing defaults than `lattice`.
- Very powerful for exploratory data analysis.

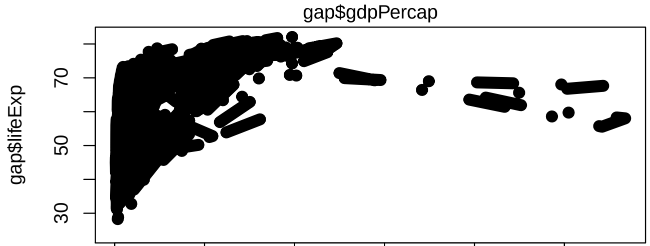

gap$lifeExp
30
50
70
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
gap$gdpPercap


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

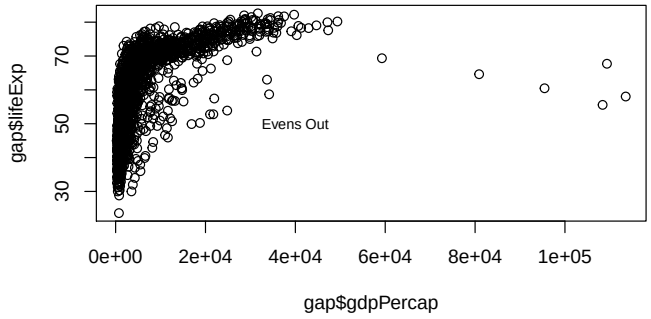


Figure 12.10:

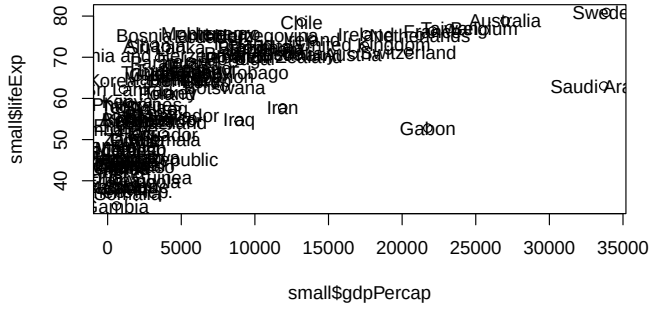


Figure 12.11:

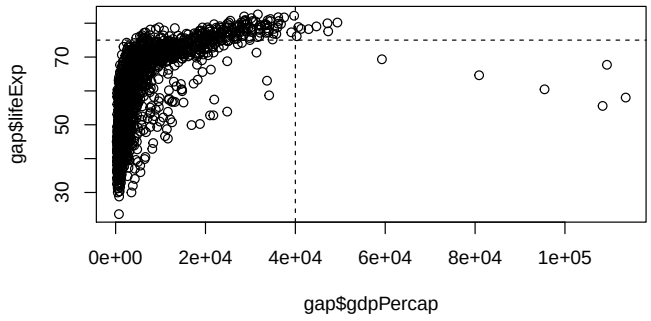


Figure 12.12:

- 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.

12.3.1 Grammar

The general call for `ggplot2` looks like this:

```
ggplot(data=, aes(x=, y=), color=, size=,) + geom_xxxx()+geom_yyyy()
```

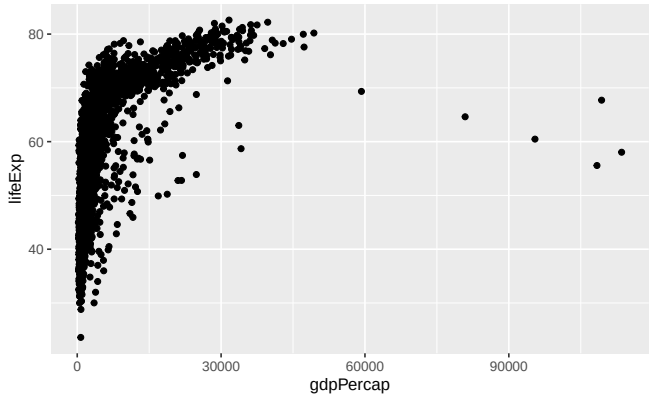
The *grammar* involves some basic components:

1. **Data**: A dataframe.
2. **Aes**thetics: How your data are represented visually, i.e., its "mapping." Which variables are shown on the x- and 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 are 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 have passed in two arguments to `ggplot`. First, we told `ggplot` what `data` we wanted 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 wanted to plot the lifeExp column of the gapminder dataframe on the x-axis, and the gdpPercap column on the y-axis.

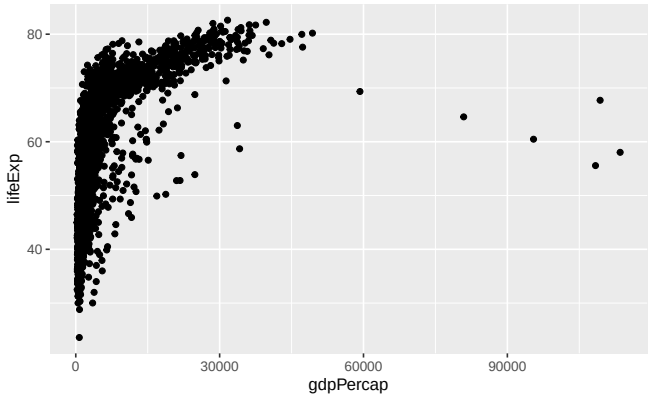
Notice that we did not need to explicitly pass aes these columns (e.g., x = gapminder$lifeExp). This is because ggplot is smart enough to know to look in the data for that column!

By itself, the call to ggplot is not 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.

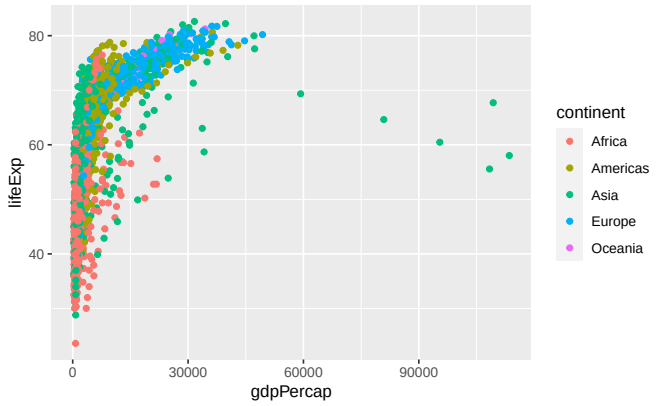
Modify the example so that the figure visualizes how life expectancy has changed over time.

Hint: The gapminder dataset has a column called year, which should appear on the x-axis.

12.3.2 Anatomy of `aes`

In the previous examples and challenge, we have 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.

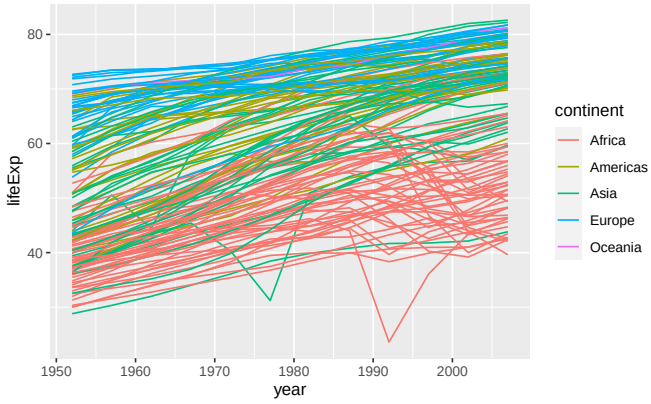
Color is not the only aesthetic argument we can set to display variation in the data. We can also vary by shape, size, etc.

```
ggplot(data=, aes(x=, y=, by =, color=, linetype=, shape=, size=))
```

12.3.3 Layers

In the previous challenge, you plotted `lifExp` over time. Using a scatterplot probably is not the best for visualizing change over time. Instead, let's tell `ggplot` to visualize the data as a line plot:

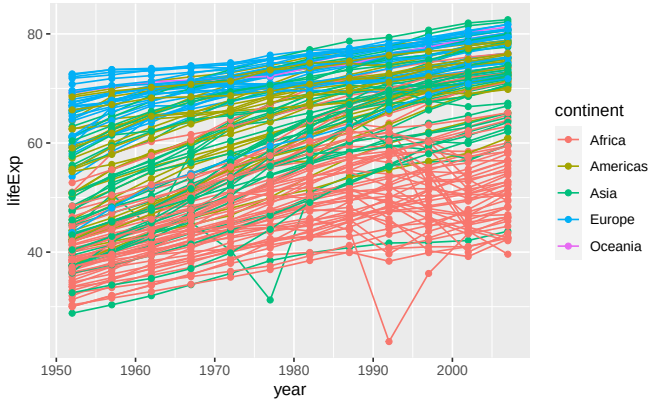
```
ggplot(data = gap, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line()
```



Instead of adding a `geom_point` layer, we have added a `geom_line` layer. We have also added the `**by**` aesthetic, which tells `ggplot` to draw a line for each country.

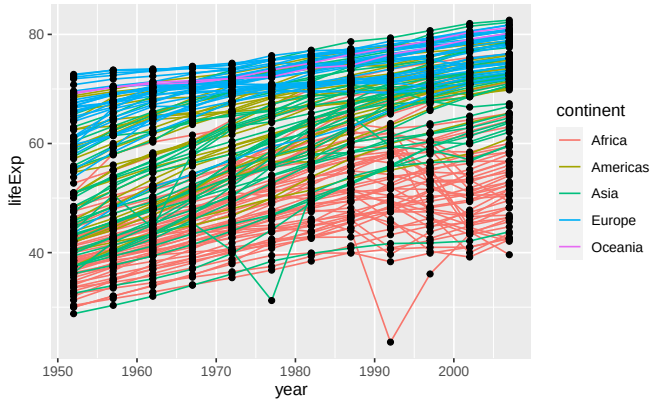
But what if we want to visualize both lines and points on the plot? We can simply add another layer to the plot:

```
ggplot(data = gap, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line() +
  geom_point()
```



It is 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 is 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.

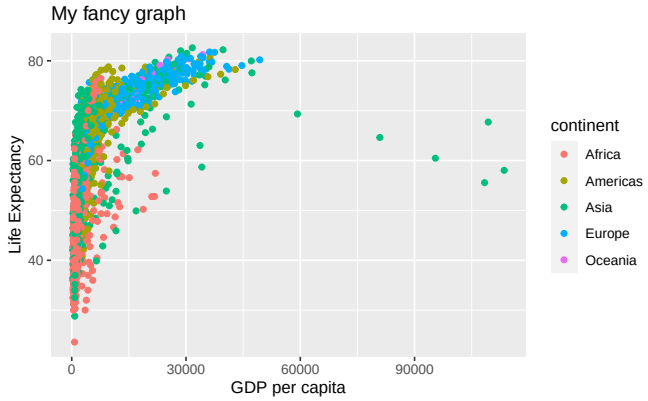
Challenge 2.

Switch the order of the point and line layers from the previous example. What happened?

12.3.4 Labels

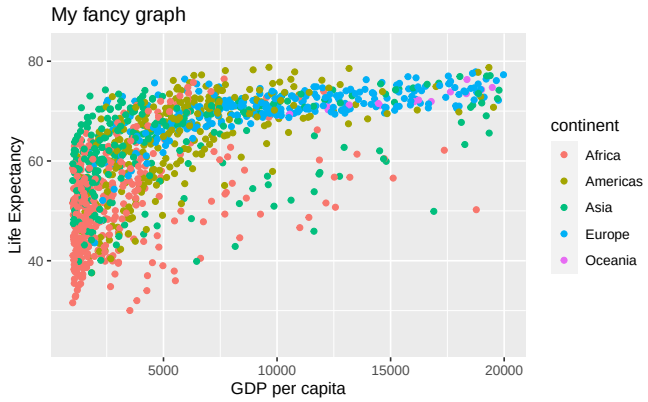
Labels are considered to be their own layers in `ggplot`.

```
# 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:

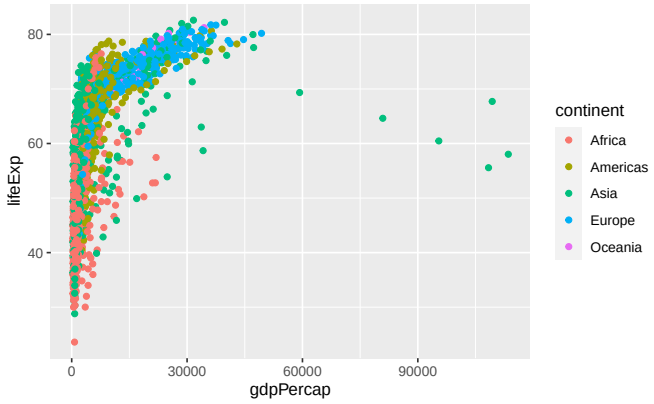
```
# Limit x-axis from 1,000 to 20,000
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  xlab("GDP per capita") +
  ylab("Life Expectancy") +
  ggtitle("My fancy graph") +
  xlim(1000, 20000)
#> Warning: Removed 515 rows containing missing values (geom_point).
```



12.3.5 Transformations and Stats

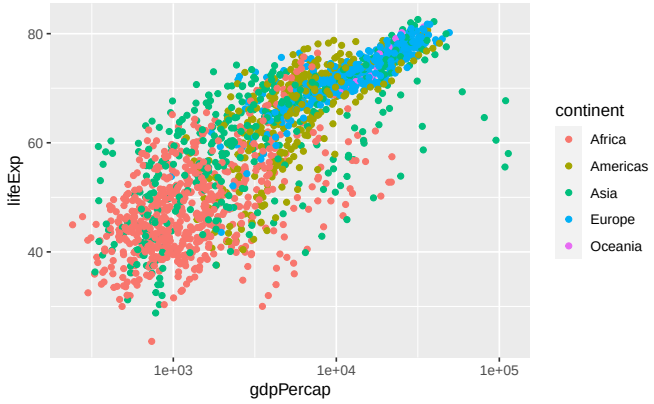
`ggplot` also makes it easy to overlay statistical models over the data. To demonstrate, we will go back to an earlier example:

```
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, which control the mapping between the data values and visual values of an aesthetic.

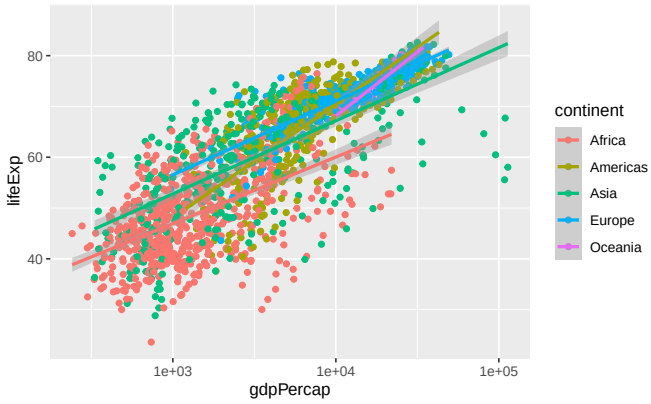
```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  scale_x_log10()
```



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 visualize the spread of data on the x-axis.

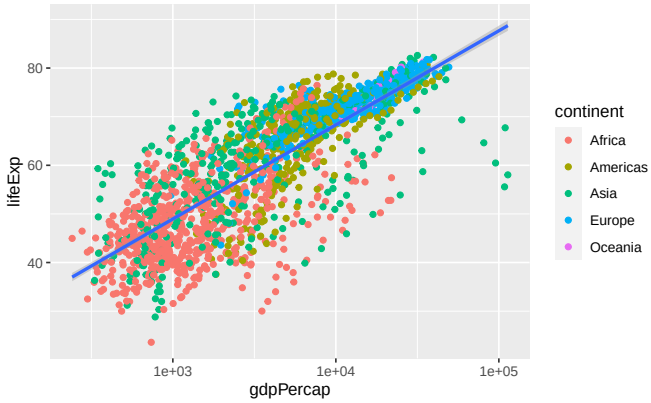
We can fit a simple relationship to the data by adding another layer, `geom_smooth`:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp, color=continent)) +
  geom_point() +
  scale_x_log10() +
  geom_smooth(method="lm")
#> `geom_smooth()` using formula 'y ~ x'
```



Note that we have 5 lines, one for each region, because of the **color** option in 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")
#> `geom_smooth()` using formula 'y ~ x'
```

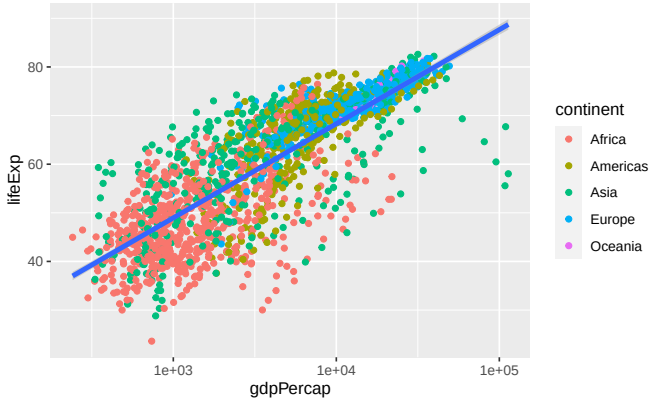


So, there are two ways an aesthetic can be specified. Here, we set the **color** aesthetic by passing it as an argument to **geom_point**. Previously in the lesson, we used the **aes** function to define a *mapping* between data variables and their visual representation.

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)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



Challenge 3.

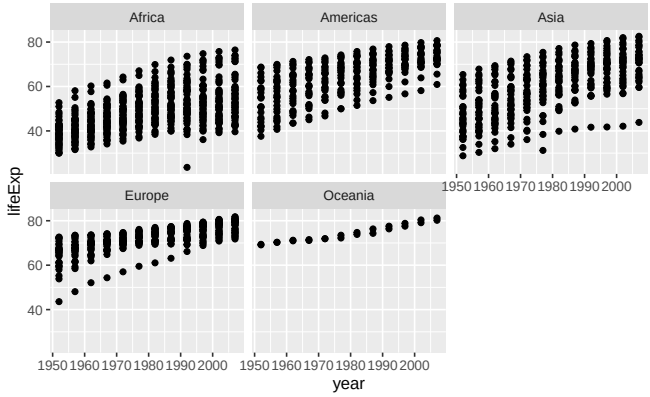
Modify the color and size of the points on the point layer in the previous example so that they are fixed (i.e., not reflective of continent).

Hint: Do not use the `aes` function.

12.3.6 Facets

Earlier, we visualized 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)
```

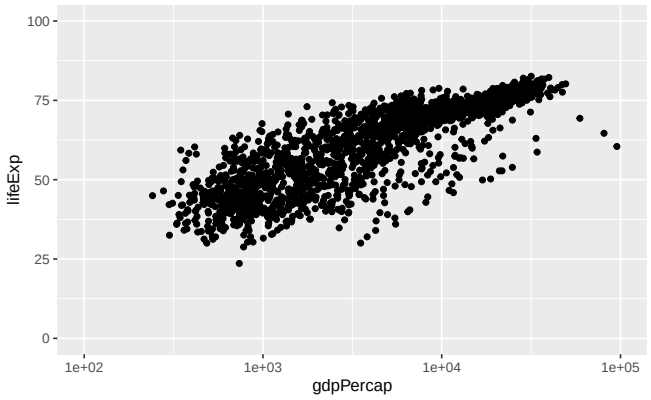


12.3.7 Legend and Scale Manipulations

When creating plots with `ggplot`, you will notice that legends are sometimes automatically produced. Additionally, you will often need to transform the axis scales, similar to the modifcations we made with the base plots.

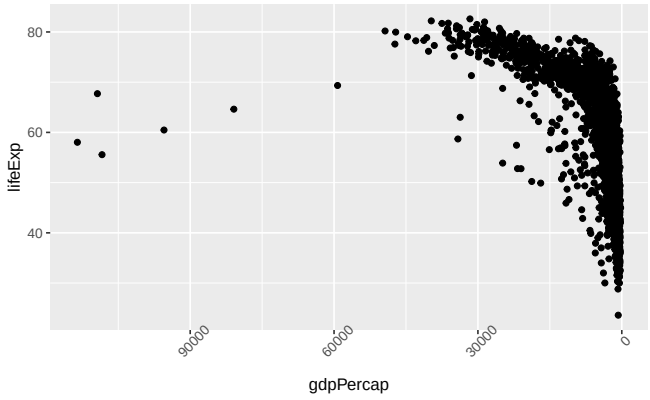
We can easily set axis limits with `xlim` and `ylim` layers, or with the `limits` argument withing `scale_x_log10`:

```
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_log10(limits = c(100, 100000)) +
  ylim(0, 100)
#> Warning: Removed 3 rows containing missing values (geom_point).
```



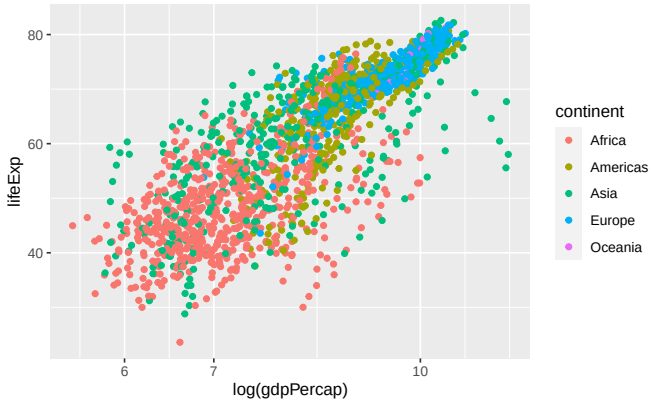
There are many other axis features we can change. For example, we can change the angle of an axis text with `theme` or reverse the direction of an axis with `scale_x_reverse` or `scale_y_reverse`. Stack Overflow is a great resource for a variety of axis transformations.

```
library(scales)
#>
#> Attaching package: 'scales'
#> The following object is masked from 'package:purrr':
#>
#>     discard
#> The following object is masked from 'package:readr':
#>
#>     col_factor
ggplot(data = gap, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  theme(axis.text.x = element_text(angle=45)) +
  scale_x_reverse()
```



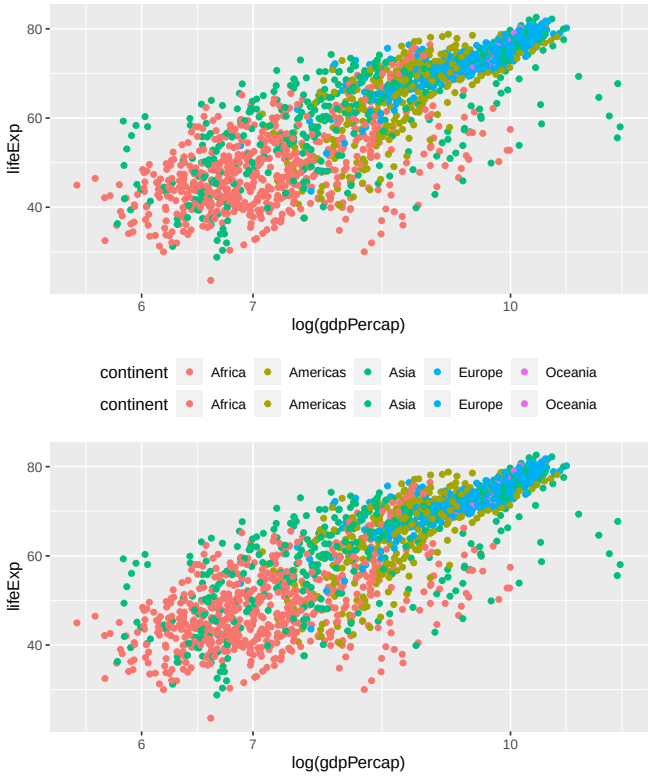
Legend manipulations can be a little trickier. Let's consider a plot where we group our observations by continent:

```
legend_plot <- ggplot(data = gap, aes(x = log(gdpPercap), y = lifeExp, color=continent)
  geom_point() +
  scale_x_log10()

legend_plot
```



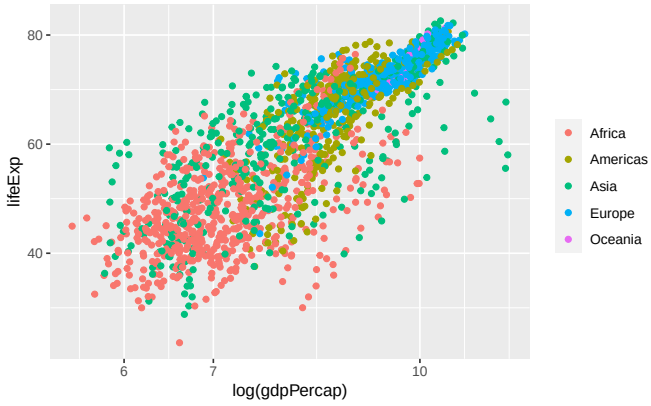
First, let's change the legend position:

```
legend_plot +
  theme(legend.position="bottom")

legend_plot +
  theme(legend.position="top")
```



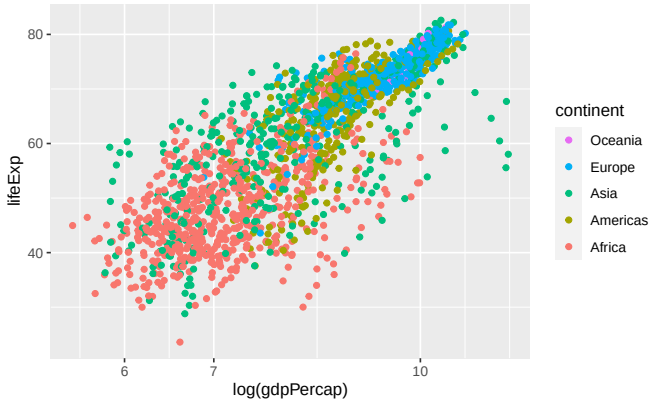
We can also remove the title of the legend for self-explanatory groupings:

```
legend_plot +
  theme(legend.title = element_blank())
```



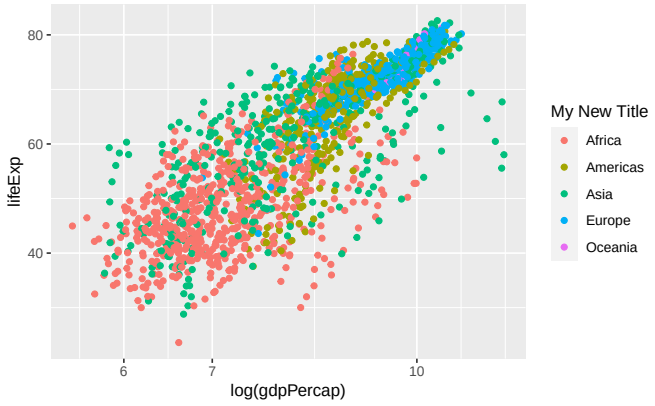
We can reverse the order of the groups with the `guides` layer. Here, we specify that we want to reverse `color`, because that was the original aesthetic that we specified to create the groupings and the legend:

```
legend_plot +
  guides(color = guide_legend(reverse = TRUE))
```



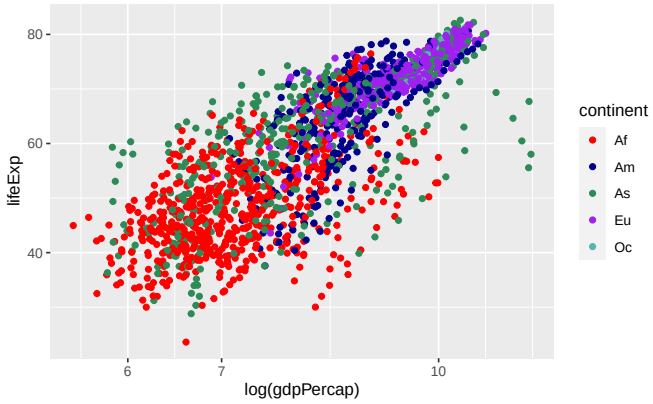
Finally, we can also change the legend title by modifying the label for `color`:

```
legend_plot +
  labs(color = "My New Title")
```



And we can change the legend labels and colors. Again, we are using `scale_color_manual` because we originally specified the groups with the `**color**` aesthetic:

```
legend_plot +
  scale_color_manual(labels = c("Af", "Am", "As", "Eu", "Oc"),
                     values = c("red", "darkblue", "seagreen", "purple", "#5ab4ac"))
```

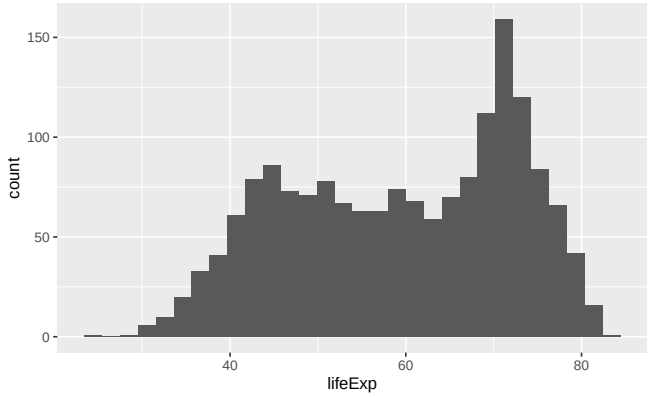


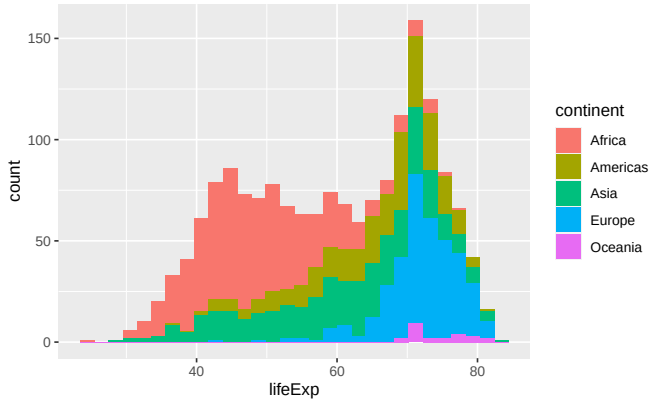
12.3.8 Putting Everything Together

Here are some other common `geom` layers:

Bar Plots

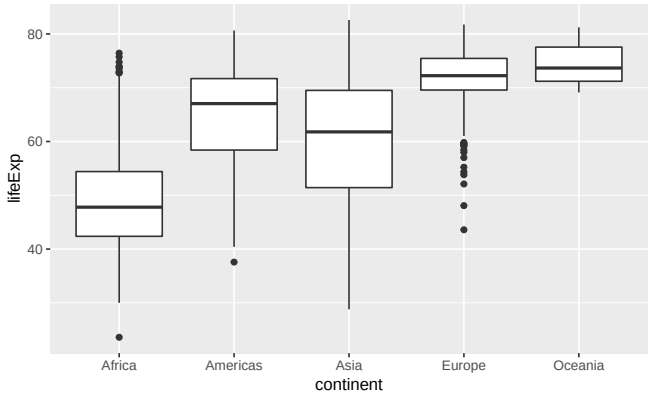
```
# Count of lifeExp bins
ggplot(data = gap, aes(x = lifeExp)) +
  geom_bar(stat="bin")
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

# 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 visualize the data spread.
- Add a facet layer to panel the density plots by year.

12.4 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 as an object called example.plot
ggsave(filename="example.pdf", plot=example.plot, scale=, width=, height=)
```

Chapter 13

Statistical Inferences

```
# setup
library(gapminder)
gap <- gapminder
```

13.1 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()`

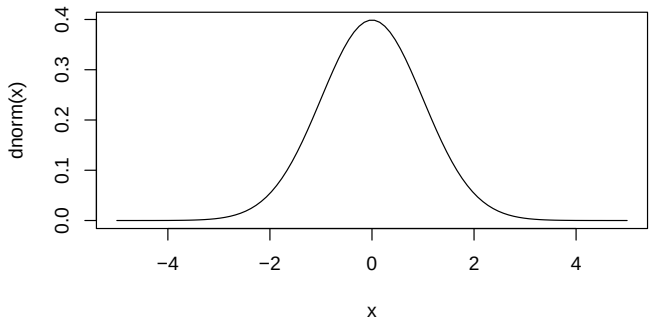
Let's see some of these functions in action with the normal distribution (mean 0, standard deviation 1):

```
dnorm(1.96) # Probability density of 1.96 from the 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) # Cumulative 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 distribution
- `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')
```



13.1.1 Sampling and Simulation

We can draw a sample with or without replacement with `sample`.

```
sample(1:nrow(gap), 20, replace = FALSE)
#>  [1]  447 1284  752  674 1699 1241  726 1579 1568 1094  265  150 1023 1290 1440
#> [16]  117  930 1553  180 1208
```

`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] 59.4
```

13.1.2 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] 1017  679  129  930 1533  471  299  270 1211 1331

vals <- sample(1:nrow(gap), 10)
vals
#>  [1]  597 1301 1518  330 1615   37 1129  729  878  485

set.seed(1)
vals <- sample(1:nrow(gap), 10)
vals
#>  [1] 1017  679  129  930 1533  471  299  270 1211 1331
```

13.1.3 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 is 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?

13.2 Inferences and Regressions

Once we have 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 hypothesis tests
2) Estimating regression models

```
# Setup
library(gapminder)
library(tidyverse)
library(broom)

# read gapminder
gap <- gapminder
```

13.2.1 Statistical Tests

Let's say we are interested in whether the life expectancy in 1967 is different than in 1977.

NB: `pull()` is a `dplyr` method to select a single variable as store it as a 1d vector.

```
# Pull out life expectancy by different years
lifeExp_1967 <- gap %>%
  filter(year==1967) %>%
  pull(lifeExp)

lifeExp_1977 <- gap %>%
  filter(year==1977) %>%
  pull(lifeExp)
```

One can test for differences in distributions in either:

1) Their means using t-tests:

```
# t test of means
t.test(x = lifeExp_1967, y = lifeExp_1977)
#>
#>  Welch Two Sample t-test
#>
#> data:  lifeExp_1967 and lifeExp_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 = lifeExp_1967, y = lifeExp_1977)
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#> data:  lifeExp_1967 and lifeExp_1977
#> D = 0.2, p-value = 0.008
#> alternative hypothesis: two-sided
```

13.2.2 Regressions and Linear Models

Running regressions in R is generally straightforward. There are two basic, catch-all regression functions in R:

- `lm` fits a standard linear regression with OLS (equivalent to `glm` with `family = gaussian(link = "identity")`).
- `glm` fits a generalized linear model with your choice of family/link function.

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")`
- `poisson(link = "log")`.

Formulas

We specify the statistical relationships to fit using a *formula*. The variable on the left-hand side of a tilde (`~`) is called the "dependent variable", while the

variables on the right-hand side are called the "independent variables" and are joined by plus signs + (for a basic linear combination).

Example: Suppose we want to regress life expectancy on GDP per capita and population, as well as continent and year.

The `lm` call looks something like this:

```
mod <- lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent + year,
          data = gap)
```

The `glm` call would look something like this:

```
mod <- glm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent + year,
           data = gap,
           family = gaussian(link = "identity"))
```

In addition to the + symbol, there are also other operators to control your formulas:

- for removing terms; : for interaction; * for crossing; . for all other variables in the dataframe that haven't yet been included in the model.

Let's see some of these in action:

```
# `.` includes all  variables but `-` removes specific terms:
mod.1 <- lm(lifeExp ~ . - country,
            data = gap)
summary(mod.1)
#>
#> Call:
#> lm(formula = lifeExp ~ . - country, data = gap)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -28.405  -4.055   0.232   4.507  20.022
#>
#> Coefficients:
#>                     Estimate Std. Error t value Pr(>|t|)
#> (Intercept)       -5.18e+02   1.99e+01   -26.1   <2e-16 ***
#> continentAmericas  1.43e+01   4.95e-01    28.9   <2e-16 ***
#> continentAsia      9.38e+00   4.72e-01    19.9   <2e-16 ***
#> continentEurope    1.94e+01   5.18e-01    37.4   <2e-16 ***
#> continentOceania   2.06e+01   1.47e+00    14.0   <2e-16 ***
#> year               2.86e-01   1.01e-02    28.5   <2e-16 ***
#> pop                1.79e-09   1.63e-09     1.1     0.27
#> gdpPercap          2.98e-04   2.00e-05    14.9   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
#>
#> Residual standard error: 6.88 on 1696 degrees of freedom
#> Multiple R-squared:  0.717,  Adjusted R-squared:  0.716
#> F-statistic:  614 on 7 and 1696 DF,  p-value: <2e-16

# `x1:x2` interacts all terms in `x1` with all terms in `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: (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
```

```
# `x1*x2` produces the cross of `x1` and `x2`, or `x1 + x2 + x1:x2`:
mod.3 <- lm(lifeExp ~ log(gdpPercap) + log(pop) + continent*factor(year),
            data = gap)
summary(mod.3)
#>
#> 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
```

Categorical variables (Factors)

Note that we wrapped the `year` variable 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 are not factorized, you can tell `lm/glm` to factorize a variable (i.e., create dummy variables on the fly) by writing `factor()`.

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.

13.2.3 Regression Output

When we store this regression in an object, we get access to several items of interest.

First, R has a helpful `summary` method for regression objects:

```
summary(mod)
#>
#> 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 specific information from the model object itself:

1. All components contained in the regression output:

```
names(mod)
#>  [1] "coefficients"  "residuals"     "effects"       "rank"
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"
#>  [9] "contrasts"     "xlevels"       "call"          "terms"
#> [13] "model"
```

2. Regression coefficients:

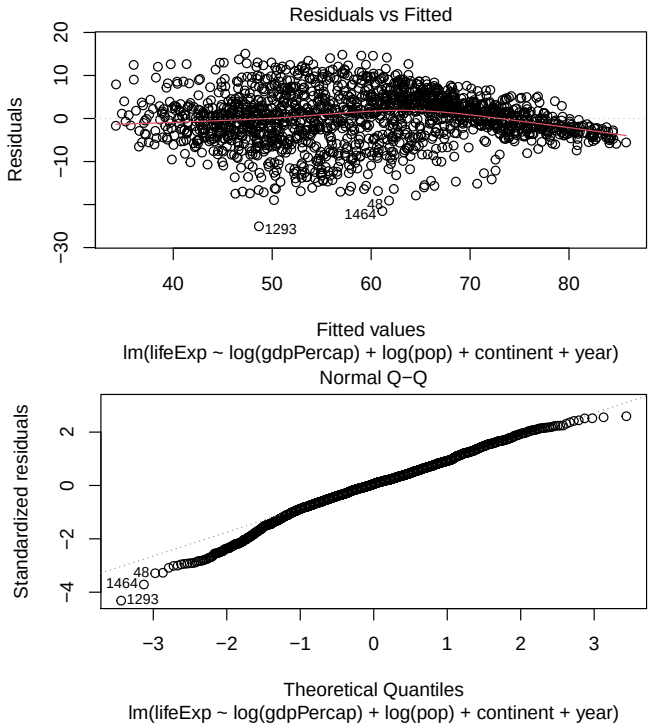
```
mod$coefficients
#>       (Intercept)    log(gdpPercap)          log(pop) continentAmericas
#>          -460.813             5.076             0.153             8.745
#>     continentAsia   continentEurope  continentOceania              year
#>             6.825            12.281            12.540             0.238
```

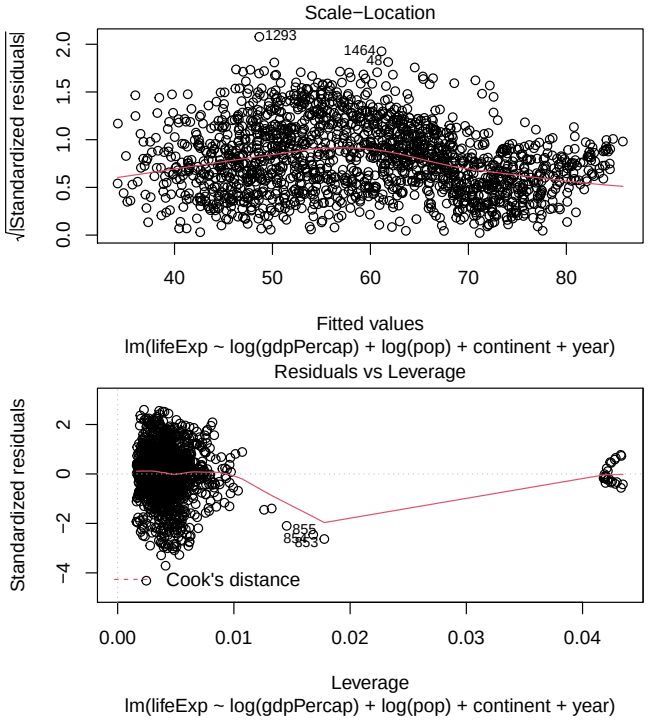
3. Regression degrees of freedom:

```
mod$df.residual
#> [1] 1696
```

4. Standard (diagnostic) plots for a regression:

```
plot(mod)
```





13.2.4 Tidying model data with `broom`

The `broom` package, by David Robinson, turn models into tidy data. `broom::tidy(model)` returns a row for each coefficient in the model. Each column gives information about the estimate or its variability.

```
mod_tidy <- tidy(mod)
mod_tidy
#> # A tibble: 8 x 5
#>   term              estimate std.error statistic   p.value
#>   <chr>                <dbl>     <dbl>     <dbl>     <dbl>
#> 1 (Intercept)       -461.      17.0       -27.2  3.96e-135
#> 2 log(gdpPercap)       5.08     0.163      31.2  3.37e-169
#> 3 log(pop)             0.153    0.0967      1.58 1.14e-  1
#> 4 continentAmericas    8.75     0.477      18.3  9.61e- 69
#> 5 continentAsia        6.83     0.423      16.1  1.49e- 54
#> 6 continentEurope     12.3      0.529      23.2  1.12e-103
#> # ... with 2 more rows
```

This is a powerful technique for working with large numbers of models because once you have tidy data, you can apply all of the techniques that you've learned about earlier in the book.

13.2.5 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
#> ================================================
```

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)
#> --------------------------------------------------------------------------------
#> log(gdpPercap)               8.340***                6.590***                5
#>                               (0.143)                 (0.182)
#>
#> log(pop)                     1.280***                0.866***
#>                               (0.111)                 (0.111)
#>
#> continentAmericas                                    6.170***                8
#>                                                       (0.555)
#>
#> continentAsia                                        4.670***                6
#>                                                       (0.494)
#>
#> continentEurope                                      8.560***                1
#>                                                       (0.608)
```

```
#>
#> continentOceania                                    8.350***                  12.500***
#>                                                      (1.510)                   (1.280)
#>
#> year                                                                          0.238***
#>                                                                               (0.009)
#>
#> Constant                          -28.800***          -12.000***               -461.000***
#>                                     (2.080)             (2.270)                  (17.000)
#>
#> ------------------------------------------------------------------------------------------
#> Observations                         1,704               1,704                    1,704
#> R2                                   0.677               0.714                    0.798
#> Adjusted R2                          0.677               0.713                    0.798
#> Residual Std. Error          7.340 (df = 1701)   6.920 (df = 1697)        5.810 (df = 1696
#> F Statistic         1,786.000*** (df = 2; 1701) 707.000*** (df = 6; 1697) 960.000*** (df = 7;
#> ==========================================================================================
#> Note:                                                        *p<0.1; **p<0.05; ***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 the 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", "Americas", "Asia
          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 are 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", "Ameri
          omit = "Constant",
          keep.stat="n", style = "ajps")
#>
#> % Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: h
#> % Date and time: Tue, Oct 20, 2020 - 15:46:34
#> \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", "Americas", "Asia
          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", "Americas", "Asia
          omit = "Constant",
          keep.stat="n", style = "ajps",
          out = "regression-table.html")
```

13.2.6 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).

Challenge 4

Check out the program below. For each line of code, write a comment describing what it does.

NB: You might some see some code we haven't yet covered in class. Try to guess what it does anyway.

```
mod <- lm(lifeExp ~ country + year + log(pop) + log(gdpPercap), data = gap)

mod_df <- tidy(mod) %>%
  slice(2:142) %>%
  separate(term, into = c("term", "country"), sep = "country") %>%
  select(country, estimate, std.error)

mod_df <- mod_df %>%
  mutate(low = estimate - qnorm((1-0.95)/2)*(std.error),
         hi = estimate + qnorm((1-0.95)/2)*(std.error))

ggplot(mod_df, aes(x =  fct_reorder(country, estimate),
                   y = estimate,
                   ymin = low,
                   ymax = hi)) +
  geom_pointrange(colour = "red", size = .2) +
  ylab("Marginal Effect on Life Expectancy") +
```

```
  xlab("Country") +
  coord_flip() +
  theme(axis.text.y = element_text(size = 5))
```

Chapter 14

Data Classes and Structures

To make the best use of the R language, you will 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 organized by their dimensionality (1d, 2d, or nd) and whether they are 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 how they work.

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

14.1 Vectors

Your garden variety R object is a vector. Vectors are 1-dimensional chains of values. We call each value an *element* of a vector.

14.1.1 Creating Vectors

A single piece of information that you regard as a scalar is just a vector of length 1. R will cheerfully let you add stuff to it with `c()`, which is short for 'combine':

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

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

# Other ways to make a vector
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
```

14.1.2 Vectors Are Everywhere

R is built to work with vectors. Many operations are vectorized, meaning they will perform calculations on each component by default. Novices often do not internalize or exploit this and they write lots of unnecessary for loops.

```
a <- c(1, -2, 3)
a^2
#> [1] 1 4 9
```

We can also add 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)
```

When reading function documentation, keep your eyes peeled for arguments that can be vectors. You will be surprised how common they are. For example, the mean of random normal variables can be provided as a vector.

```
set.seed(1999)
rnorm(5, mean = c(10, 100, 1000, 10000, 100000))
#> [1]     10.7    100.0   1001.2  10001.5 100000.1
```

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

14.1.3 Recycling

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

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

```
(y <- 1:10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
(z <- 3:7)
#> [1] 3 4 5 6 7
y + z
#>  [1]  4  6  8 10 12  9 11 13 15 17
```

14.1.4 Types of Vectors

There are four common types of vectors, depending on the class:

1. `integer`
2. `numeric` (same as `double`)
3. `character`
4. `logical`

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"
```

Logical (Boolean) Vectors

Logical vectors take on one of three possible values:

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

They are often used in conjunction with Boolean expressions.

```
b1 <- c(TRUE, TRUE, FALSE, NA)
b1
#> [1]  TRUE  TRUE FALSE    NA

vec_1 <- c(1, 2, 3)
vec_2 <- c(1, 9, 3)
vec_1 == vec_2
#> [1]  TRUE FALSE  TRUE

b2 <- vec_1 == vec_2
```

14.1.5 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()`.

```
vec_1 <- c(1, 2, 3)
vec_2 <- c(1, 9, 3)
boo_1 <- vec_1 == vec_2
```

```
# Total number of TRUEs
sum(boo_1)
#> [1] 2

# Proportion that are TRUE
mean(boo_1)
#> [1] 0.667
```

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, will not 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
```

14.1.6 Naming a Vector

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

You can give names 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",

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 do not have to do this when creating a named vector.

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

14.1.7 Challenges

Challenge 1: Create and examine your vector.

Create a character vector called `fruit` that contains 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 from 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 to 20 and multiply it by the original vector from Challenge 2.

14.2 Subsetting Vectors

Sometimes we want to isolate elements of a vector for inspection, modification, etc. This is often called **indexing** or **subsetting**.

> By the way, indexing begins at 1 in R, unlike many other languages that index from 0.

14.2.1 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.

The simplest way to subset a vector is with a single integer:

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

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

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

You cannot 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["d"]
#>   d
#> 5.4
x[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1
```

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
```

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

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

This command tests if the condition stated by the comparison operator is `TRUE` or `FALSE` for every element of the vector, 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 <- c(2.1, 4.2, 3.3, 5.4)
x[x > 3]
#> [1] 4.2 3.3 5.4

# With !
!x > 5
#> [1]  TRUE  TRUE  TRUE FALSE
x[!x > 5]
#> [1] 2.1 4.2 3.3

# With %in%
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
```

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).
```

14.3 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 are not actually helpful.

14.3.1 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
```

14.3.2 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
```

14.3.3 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
```

14.3.4 Factors Are Integers

Factors are pretty much integers that have labels on them. Underneath, they are 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 is usually best to explicitly **convert** factors to character vectors if you need string-like behavior.

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

14.3.5 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))
```

14.4 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.

In data analysis, you will not make lists very often, at least not consciously, but you should still know about them. Why?

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

14.4.1 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
```

14.4.2 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 have 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")
```

14.4.3 List Structure

A very useful tool for working with lists is `str()`, because it focuses on reviewing the structure of a list, not its 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 dataframes and linear model 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 of 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
#>  $ 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
#>  $ 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"
```

14.5 Subsetting Lists

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

The "pepper shaker photos" in R for Data Science are a splendid visual explanation of the different ways to get stuff out of a list. Highly recommended.

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

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

14.5.1 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]
#> $a
#> [1] 1 2 3
str(my_list[1])
#> List of 1
#>  $ a: int [1:3] 1 2 3

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"
```

14.5.2 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."

— ?

14.5.3 with $

`$` is a shorthand for extracting a single named element of a list. It works especially well when coupled with tab completion.

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

14.5.4 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]]`?

Challenge 5.

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.

14.6 Matrices

Matrices are like 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.

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

> General arrays are available in R, where a matrix is an important special case having dimension 2.

Practically speaking, matrices are good for large tables of numbers. However, as social scientists, we rarely work with purely numerical data.

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

Let's make a simple matrix and give it decent row and column names. You will see familiar or self-explanatory functions below for getting to know a matrix.

```
## Do not worry if the construction of this matrix confuses you;
## just focus on the product
m <- outer(as.character(1:4), as.character(1:4),
           function(x, y) {
             paste0('x', x, '-', y)
             })
m
#>      [,1]   [,2]   [,3]   [,4]
```

```
#> [1,] "x1-1" "x1-2" "x1-3" "x1-4"
#> [2,] "x2-1" "x2-2" "x2-3" "x2-4"
#> [3,] "x3-1" "x3-2" "x3-3" "x3-4"
#> [4,] "x4-1" "x4-2" "x4-3" "x4-4"
str(m)
#>  chr [1:4, 1:4] "x1-1" "x2-1" "x3-1" "x4-1" "x1-2" "x2-2" "x3-2" "x4-2" ...
class(m)
#> [1] "matrix" "array"
dim(m)
#> [1] 4 4
nrow(m)
#> [1] 4
ncol(m)
#> [1] 4
rownames(m)
#> NULL

rownames(m) <- c("row1", "row2", "row3", "row4")
colnames(m) <- c("col1", "col2", "col3", "col4")

m
#>      col1   col2   col3   col4
#> row1 "x1-1" "x1-2" "x1-3" "x1-4"
#> row2 "x2-1" "x2-2" "x2-3" "x2-4"
#> row3 "x3-1" "x3-2" "x3-3" "x3-4"
#> row4 "x4-1" "x4-2" "x4-3" "x4-4"
```

14.7 Indexing a Matrix

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 and another for columns to select – separated by a comma. Blank subsetting is also useful because it lets you keep all rows or all columns.

```
m[2, 3] # Selects the value at the second row and third column
#> [1] "x2-3"

m[2, ] # We get row 2
#>   col1   col2   col3   col4
#> "x2-1" "x2-2" "x2-3" "x2-4"

m[ , 3, drop = FALSE] # We get column 3
#>      col3
```

```
#> row1 "x1-3"
#> row2 "x2-3"
#> row3 "x3-3"
#> row4 "x4-3"

dim(m[ , 3, drop = FALSE]) # We get column 3 as a 4 x 1 matrix
#> [1] 4 1

m[c("row1", "row4"), c("col2", "col3")] # We get rows 1, 4 and columns 2, 3
#>      col2   col3
#> row1 "x1-2" "x1-3"
#> row4 "x4-2" "x4-3"

m[-c(2, 3), c(TRUE, TRUE, FALSE, FALSE)] # Wacky but possible
#>      col1   col2
#> row1 "x1-1" "x1-2"
#> row4 "x4-1" "x4-2"
```

14.8 Dataframes

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

Hopefully the slog through vectors, matrices, and lists will be redeemed by greater prowess at manipulating `data.frames`. Why should this be true?

1. A dataframe is a *list*.
2. The list elements are the variables, and they are atomic vectors.
3. Dataframes are rectangular, like their matrix friends, so your intuition – and even some syntax – can be borrowed from the matrix world.

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 do not worry about the difference.

14.8.1 Creating Dataframes

We have already worked extensively with dataframes that we have imported through a package or `read.csv`.

```
library(gapminder)
gap <- gapminder
```

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

```
vec_1 <- 1:3
vec_2 <- c("a", "b", "c")
df <- data.frame(vec_1, vec_2)
```

14.8.2 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.

```
vec_1 <- 1:3
vec_2 <- c("a", "b", "c")
df <- data.frame(vec_1, vec_2)

str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ vec_1: int  1 2 3
#>  $ vec_2: chr  "a" "b" "c"
```

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.

```
vec_1 <- 1:3
vec_2 <- c("a", "b", "c")
df <- data.frame(vec_1, vec_2)

# 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
```

14.8.3 Naming Dataframes

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

```
vec_1 <- 1:3
vec_2 <- c("a", "b", "c")
df <- data.frame(vec_1, vec_2)

# These two are equivalent
names(df)
#> [1] "vec_1" "vec_2"
colnames(df)
#> [1] "vec_1" "vec_2"

# Change the colnames
colnames(df) <- c("Number", "Character")

# 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
```

14.9 Indexing Dataframes

> A dataframe is a list that quacks like a matrix.

Remember that dataframes are really lists of vectors (one vector per column). That means that dataframes have both list- and matrix-like behavior.

For example, just as `list$name` selects the `name` element from the list, `df$name` selects the `name` column (vector) from the dataframe:

```
library(gapminder)
gap <- gapminder

head(gap$country)
#> [1] Afghanistan Afghanistan Afghanistan Afghanistan Afghanistan Afghanistan
#> 142 Levels: Afghanistan Albania Algeria Angola Argentina Australia ... Zimbabwe
```

Likewise, we can use square brackets to subset rows and columns:

```
# Row 1, column 3
gap[1, 3]
#> # A tibble: 1 x 1
#>    year
#>   <int>
#> 1  1952

# Fourth row
gap[4, ]
#> # A tibble: 1 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1967    34.0 11537966      836.

# First two rows of the columns 1 and 5
gap[c(1,2), c(1, 5)]
#> # A tibble: 2 x 2
#>   country         pop
#>   <fct>         <int>
#> 1 Afghanistan 8425333
#> 2 Afghanistan 9240934
```

We can also use subsetting in conjunction with assignment to quickly add a column:

```
names(gap)
#> [1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
gap$new_col <- NA
head(gap)
#> # A tibble: 6 x 7
#>   country     continent  year lifeExp      pop gdpPercap new_col
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl> <lgl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779. NA
#> 2 Afghanistan Asia       1957    30.3  9240934      821. NA
#> 3 Afghanistan Asia       1962    32.0 10267083      853. NA
#> 4 Afghanistan Asia       1967    34.0 11537966      836. NA
#> 5 Afghanistan Asia       1972    36.1 13079460      740. NA
#> 6 Afghanistan Asia       1977    38.4 14880372      786. NA
```

14.9.1 Challenges

Challenge 1.

Create a 3x2 dataframe called `basket`. The first column should contain the names of 3 fruits. The second column should contain the price of those fruits. Now give your dataframe appropriate column and row names.

Challenge 2.

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

Chapter 15

Strings and Regular Expressions

This unit focuses on character (or "string") data. We will 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 do not always have textual data, so we need to load it explicitly.

```
library(tidyverse)
library(stringr)
```

15.1 String Basics

This unit focuses on character (or "string") data. We will focus on **string basics**, such as concatinating and subsettings.

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 will need to double it up: `"\\"`.

Beware that the printed representation of a string is not the same as the 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 `?"'"`.

Sometimes you will also 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 will avoid them because they can be inconsistent, which makes them hard to remember.

Instead, we will use functions from `stringr`. `stringr` contains functions with 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 are 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, sep = ", ") # This will not work
#> [1] "x" "y" "z"
str_c(x, collapse = ", ") # But this will
#> [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 GOAT")
str_sub(x, 1, 8)
#> [1] "Rochelle"
```

```
# Negative numbers count backwards from the end
str_sub(x, -8, -1)
#> [1] "the GOAT"
```

You can also use the assignment form of `str_sub()` to modify strings:

```
x <- c("Rochelle is the GOAT")
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "rochelle is the GOAT"
```

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 each of 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 do not 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 is affected by the locale is sorting. The base R `order()` and `sort()` functions sort strings using the current locale. If you want robust behavior 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"
```

15.1.7 Challenges

Challenge 1.

In code that does not use `stringr`, you will often see `paste()` and `paste0()`. What is the difference between the two functions? What `stringr` function are they equivalent to? How do the functions differ in their handling of `NA`?

Challenge 2.

In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()`.

Challenge 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?

Challenge 4.

What does `str_trim()` do? What is the opposite of `str_trim()`?

```
library(tidyverse)
library(stringr)
```

15.2 Regular Expressions

Regular expressions are a very terse language that allows you to describe patterns in strings. They take a little while to get your head around, but once you understand them, you will find them extremely useful.

To learn regular expressions, we will use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression and show you how they match. We will start with very simple regular expressions and then gradually get more and more complicated. Once you have mastered pattern matching, you will 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
```

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 behavior.

Regexps use the backslash, `\`, to escape special behavior. 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 will write the 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 is 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 have 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 that, to create a regular expression containing `\d` or `\s`, you will need to escape the `\` for the string, so you will 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: `$`, `.`, `|`, `?`, `*`, `+`, `(`, `)`, `[`, and `{`. 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")
```

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.

15.2.7 Challenges

Create regular expressions to find all words that

1. Start with a vowel.
2. Only contain consonants. (Hint: Think about matching "not"-vowels.)
3. End with `ed`, but not with `eed`.
4. End with `ing` or `ise`.

```
library(tidyverse)
library(stringr)
```

15.3 Common Tools

Now that you have learned the basics of regular expressions, it is time to learn how to apply them to real problems. In this section, you will 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:

```
words<- stringr::words
# See common 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 will 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 is 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
```

Challenge 1.

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 are going to need a more complicated example. I am 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"    ""
```

Challenge 2.

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 them up by character, line, sentence, or 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"
```

Challenge 3.

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 is a string, it is 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"
```

Acknowledgments

This page was adapted from the following source:

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

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

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 is 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 is 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 <- gapminder
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.
```

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: Problem with `mutate()` input `life.35`.
#> i the condition has length > 1 and only the first element will be used
#> i Input `life.35` is `if (...) NULL`.
#> Warning in if (lifeExp >= 35) {: the condition has length > 1 and only the first
#> element will be used

head(gap_if)
#> # A tibble: 6 x 7
#>   country     continent  year lifeExp      pop gdpPercap life.35
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>   <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.       0
#> 2 Afghanistan Asia       1957    30.3  9240934      821.       0
#> 3 Afghanistan Asia       1962    32.0 10267083      853.       0
#> 4 Afghanistan Asia       1967    34.0 11537966      836.       0
#> 5 Afghanistan Asia       1972    36.1 13079460      740.       0
#> 6 Afghanistan Asia       1977    38.4 14880372      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))

gap_ifelse
#> # A tibble: 1,704 x 7
#>    country     continent  year lifeExp      pop gdpPercap life.35
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>   <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.       0
#> 2 Afghanistan Asia       1957    30.3  9240934      821.       0
#> 3 Afghanistan Asia       1962    32.0 10267083      853.       0
#> 4 Afghanistan Asia       1967    34.0 11537966      836.       0
#> 5 Afghanistan Asia       1972    36.1 13079460      740.       1
#> 6 Afghanistan Asia       1977    38.4 14880372      786.       1
#> # ... with 1,698 more rows
```

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

16.2 Functions

Functions are the basic building blocks of programs. Think of them as "mini-scripts" or "tiny commands." We have already used dozens of functions created by others (e.g., `filter()` and `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 is 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:

```
gap <- gapminder

gap_norm <- gap %>%
  mutate(pop_norm = (pop - min(pop)) / (max(pop) - min (pop)),
         gdp_norm = (gdpPercap - min(gdpPercap)) / (max(gdpPercap) - min (gdpPercap)),
         life_norm = (lifeExp - min(lifeExp) / (max(pop)) - min (lifeExp)))

summary(gap_norm$pop_norm)
```

You might be able to puzzle out that this rescales each numeric column to have a range from 0 to 1. But did you spot the mistakes? I made two errors when copying-and-pasting the code for `lifeExp`.

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 it 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?

```
# The corrected code
gap <- gapminder

gap_norm <- gap %>%
```

```
  mutate(pop_norm = (pop - min(pop)) / (max(pop) - min (pop)),
         gdp_norm = (gdpPercap - min(gdpPercap)) / (max(gdpPercap) - min (gdpPercap)),
         life_norm = (lifeExp - min(lifeExp)) / (max(lifeExp) - min (lifeExp)))

# Focus on the line
# pop_norm = (pop - min(pop)) / (max(pop) - min (pop))
```

This code only has one input: `gap$pop`. To make the inputs more clear, it is a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, which I will call `x`:

```
x <- gap$pop

(x - min(x)) / (max(x) - min(x))
```

There is still some duplication in this code. We are calulating some version of the range three times. Pulling out intermediate calculations into named variables is a good practice, because it becomes clearer what the code is doing.

```
x <- gap$pop

rng <- range(x)
(x - rng[1]) / (rng[2] - rng[1])
```

Now that I have simplified the code and checked that it still works, I can turn it into a function:

```
rescale01 <- function(x) {
  rng <- range(x)
  scales <- (x - rng[1]) / (rng[2] - rng[1])
  return(scales)
}
```

Note the overall process: I only made the function after I had figured out how to make it work with a simple input. It is easier to start with working code and turn it into a function; it is harder to create a function and then try to make it work.

At this point, it is 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 are 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:

```
rescale01 <- function(x) {
  rng <- range(x)
  scales <- (x - rng[1]) / (rng[2] - rng[1])
  return(scales)
}

gap_norm <- gap %>%
  mutate(pop_norm = rescale01(pop),
         gdp_norm = rescale01(gdpPercap),
         life_norm = rescale01(lifeExp))
```

Compared to the original, this code is easier to understand, and we have eliminated one class of copy-and-paste errors. There is still quite a bit of duplication, since we are doing the same thing to multiple columns. We will 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 have extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = T)
  scales <- (x - rng[1]) / (rng[2] - rng[1])
  return(scales)
}

rescale01(c(1, 2, NA, 3, 4, 5))
#> [1] 0.00 0.25   NA 0.50 0.75 1.00
```

16.2.5 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 and returns the total number of NAs in both vectors.

For instance, it should generate the following output:

```
vec1 <- c(NA, 4, 6, 2, NA, 5, NA)
vec2 <- c(NA, "Dec", "Apr", NA, "Jul", "Apr")

my_other_function(vec1, vec2)
# [1] 5

# Hints
is.na(vec1)
sum(c(T, F))
```

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)
}
```

Why do the two lines of code below return different values?

```
print_date(day=1, month=2, year=2003)
print_date(1, 2, 2003)
```

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 will learn about three approaches to iteratation:

1. Vectorized functions.
2. `map` and functional programming.
3. 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 Functional Programming and `map`

You might have used for loops in other languages. Loops are not as important in R as they are in other languages, because R is a **functional** programming language. This means that it is 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 (part of `tidyverse`) provides a family of functions to do it for you. They effectively eliminate the need for many common `for` loops.

```
library(tidyverse)
```

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 is 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 are wrong! (Well, at least they are rather out of date, as for loops have not 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 this simple data frame:

```
df <- data.frame(
  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
#> -0.441 -0.179 -0.124  0.152
map_dbl(df, median)
#>       a       b       c       d
#> -0.2458 -0.2873 -0.0567  0.1443
map_dbl(df, sd)
#>     a     b     c     d
#> 1.118 1.176 1.047 0.964
```

The data can even be piped!

```
df %>% map_dbl(mean)
#>      a      b      c      d
#> -0.441 -0.179 -0.124  0.152
df %>% map_dbl(median)
#>       a       b       c       d
#> -0.2458 -0.2873 -0.0567  0.1443
df %>% map_dbl(sd)
#>     a     b     c     d
#> 1.118 1.176 1.047 0.964
```

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
#> -0.2458 -0.2873 -0.0567  0.1443
```

Check out other fun applications of `map` functions here.

16.3.3 Challenges

Write code that uses one of the `map` functions to:

Challenge 1.

Calculate the arithmetic mean for every column in `mtcars`.

Challenge 2.

Calculate the number of unique values in each column of `iris`.

Challenge 3.

Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

16.3.4 Scoped Verbs

The last iteration technique we will 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 would 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 is a ton of web data that is 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.

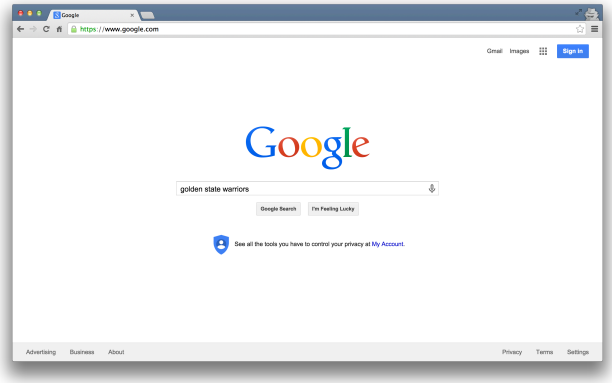
A very common type of API is the **Web API**, which (among other things) allows users to query a remote database over the internet.

Web APIs take on a variety of formats, but the vast majority adheres to a particular style known as **Representational State Transfer** or **REST**. What

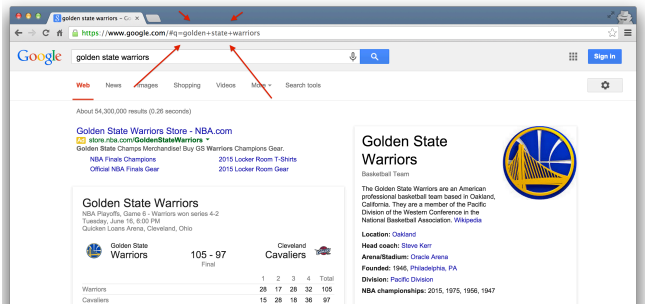
makes these "RESTful" APIs so convenient is that we can use them to query databases using URLs.

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:

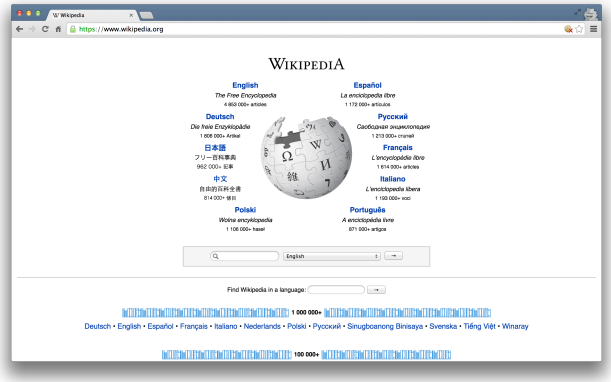
- **Uniform Resource Location (URL)**: A string of characters that, when interpreted via the Hypertext Transfer Protocol (HTTP), points to a data resource, notably files written in Hypertext Markup Language (HTML) or a subset of a database. This is often referred to as a "call."
- **HTTP Methods/Verbs**:
  - *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 rarely 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 would 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, 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 metadata (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 1,000 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 the US Census Bureau. The `tidycensus` package allows users to interface with the US Census Bureau's decennial Census and five-year American Community APIs.

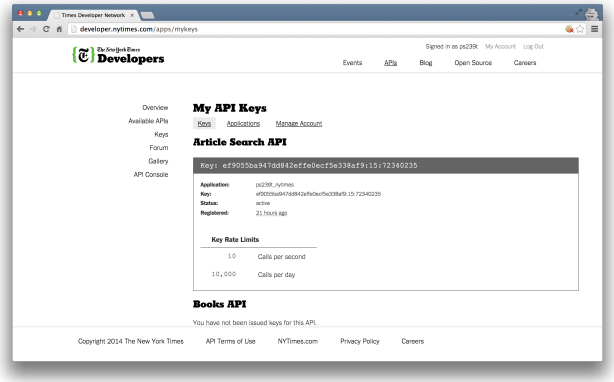
17.2.4 Getting API Access

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 will 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 have 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 are not 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.**][Collecting Twitter Data with RTweet]

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 is 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 doable.

17.3 Writing API Queries

If no wrapper package is available, we have to write our own API query and format the response ourselves using R. This is trickier, but definitely doable.

In this unit, we will 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 the 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(stringr)
library(httr)
library(jsonlite)
library(lubridate)
```

17.3.1 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. **Authentication 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 these values as variables:

```
#key <- "YOURKEY HERE"
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 do not need to configure that directly.

We are 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 are going to 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 10, you need to add a "page" parameter. Change the search parameters above to get the second 10 results.

17.3.2 Parsing the Response

We can read the content of the server's response using the `content()` function.

```
response <- httr::content(r, "text")
str_sub(response, 1, 1000)
#> [1] "{\"status\":\"OK\",\"copyright\":\"Copyright (c) 2020 The New York Times Compa
```

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 dataframe.

The `jsonlite` package provides several easy conversion functions for moving between JSON and vectors, dataframes, 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 Reserv
```

```
#>  $ response :List of 2
#>   ..$ docs:'data.frame': 10 obs. of  27 variables:
#>   ..$ meta:List of 3
```

That looks intimidating! But it is 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 10 documents:

```
# What is in 'meta'?
response_df$response$meta
#> $hits
#> [1] 1639286
#>
#> $offset
#> [1] 0
#>
#> $time
#> [1] 187

# Pull out number of hits
response_df$response$meta$hits
#> [1] 1639286

# Check out docs
names(response_df$response$docs)
#>  [1] "abstract"                "web_url"
#>  [3] "snippet"                 "lead_paragraph"
#>  [5] "source"                  "multimedia"
```

```
#>  [7] "keywords"                 "pub_date"
#>  [9] "document_type"            "news_desk"
#> [11] "section_name"             "_id"
#> [13] "word_count"               "uri"
#> [15] "type_of_material"         "print_section"
#> [17] "print_page"               "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
```

17.3.3 Iteration through Results Pager

That is 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"

    # 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 are ready to iterate over each page. First, let's review what we have

done so far:

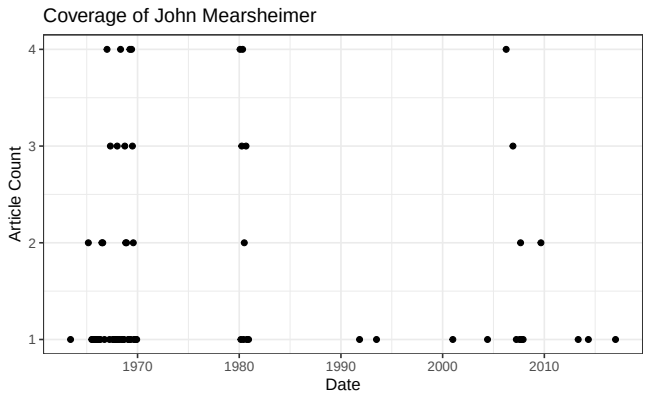
```
# Set key and base
base.url = "http://api.nytimes.com/svc/search/v2/articlesearch.json"
search_term = "John Mearsheimer" # Change me

# 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"
#> No encoding supplied: defaulting to UTF-8.
#> [1] "Scraping page:  12"
#> No encoding supplied: defaulting to UTF-8.
#> [1] "Scraping page:  13"
#> No encoding supplied: defaulting to UTF-8.
#> [1] "Scraping page:  14"
#> No encoding supplied: defaulting to UTF-8.
#> [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)
```

17.3.4 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)
#> `summarise()` ungrouping output (override with `.groups` argument)

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")
```



17.3.5 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.

17.4 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.

17.4.0.1 Some Disclaimers

- Check a site's terms and conditions before scraping.
- Be nice - do not hammer the site's server. Review these ethical webscraping tips.
- Sites change their layout all the time. Your scraper will break.

17.4.1 What Is a Website?

A website is some combination of **code**base + **data**base that lives on a server.

When it gets to us (the "front end"), it is delivered to us as HTML + CSS stylesheets + JavaScript.

```
"<!DOCTYPE html>\n<html lang=\"mul\" dir=\"ltr\">\n<head>\n<!-- Sysops:
Please do not edit the main template directly; update /temp and synchronise.
-->\n<meta charset=\"utf-8\">\n<title>Wikipedia</title>\n<!--[if lt IE 7]><meta
http-equiv=\"imagetoolbar\" content=\"no\"><![endif]-->\n<meta name=\"viewport\"
content=\"i"
```

Your browser turns that into a nice layout.

Current Senate Members *99th General Assembly*
Leadership Officers Senate Seating Chart **Democrats:** 39 **Republicans:** 20

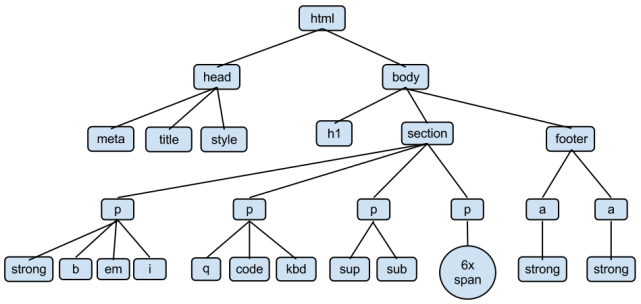
Senator	Bills	Committees	District	Party
Pamela J. Althoff	Bills	Committees	32	R
Neil Anderson	Bills	Committees	36	R
Jason A. Barickman	Bills	Committees	53	R
Scott M. Bennett	Bills	Committees	52	D
Jennifer Bertino-Tarrant	Bills	Committees	49	D
Daniel Biss	Bills	Committees	9	D
Tim Bivins	Bills	Committees	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

17.4.2 Websites Return HTML

The core of a website is **HTML** (Hyper Text Markup Language). HTML defines the **structure** of a webpage using a series of **elements**. HTML elements tell the browser how to display the page by labeling pieces of content: "This is a heading," "this is a paragraph," "this is a link," etc.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page title</title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

HTML elements can contain other elements, like a tree:



17.4.3 HTML Elements

An HTML element is defined by a start tag, some content, and an end tag.

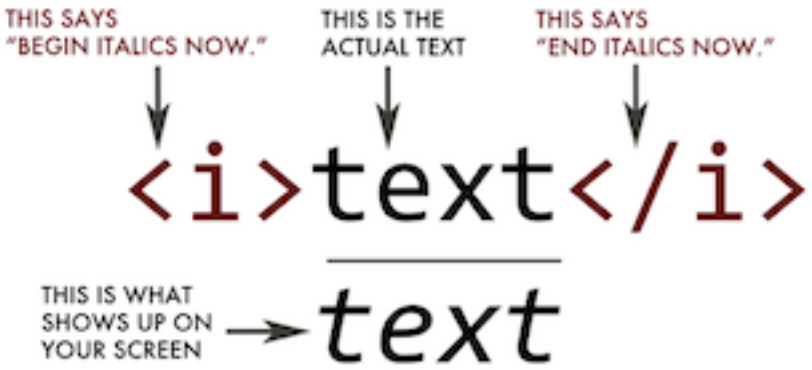


Figure 17.1: html-tags

The HTML element is everything from the start tag to the end tag.

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

17.4.3.1 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”

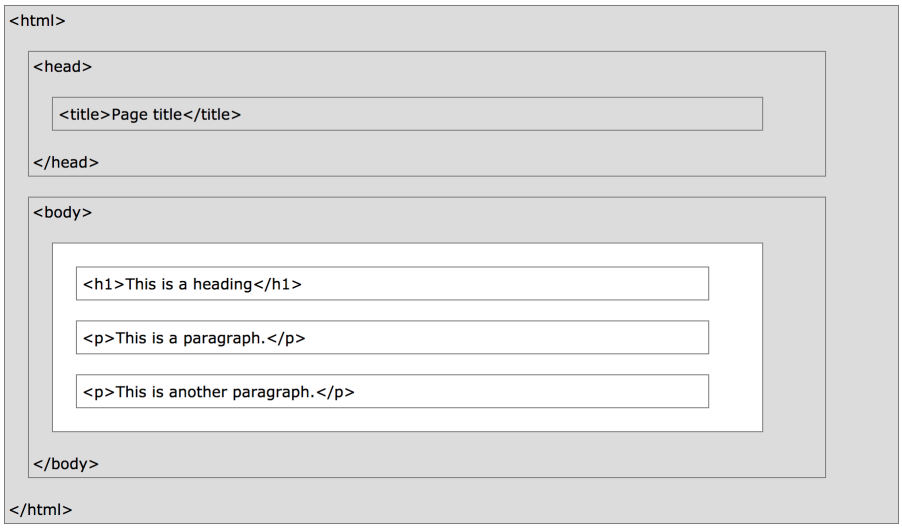
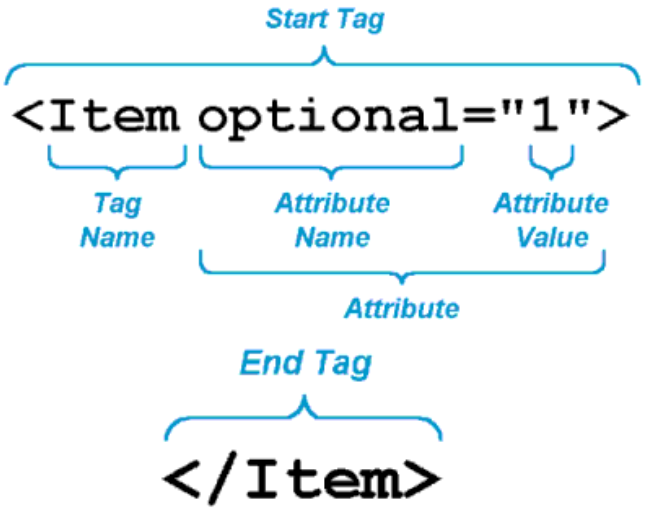


Figure 17.2: page-structure



- Sometimes we can find the data we want by just using HTML tags or attributes (e.g, all the `<a>` tags).
- More often, that is not enough: There might be 1,000 `<a>` tags on a page. But maybe we want only the `<a>` tags *inside* of a `<p>` tag.
- Enter CSS…

17.4.4 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. Some websites can easily contain 100+ individual pages, each with their own HTML code.

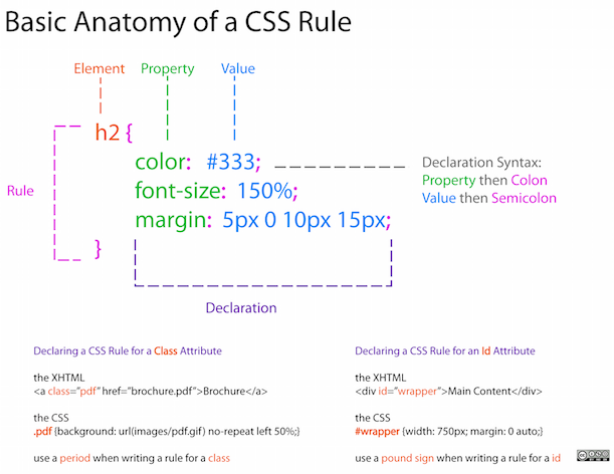
To solve this problem, CSS was created specifically to display content on a webpage. Now, one can change the look of an entire website just by changing one file.

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.

17.4.4.1 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:



17.4.4.2 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:

1. The entire table.
2. The row containing "Kurtis McCoy."
3. Just the element containing first names.

(Hint: There will be multiple solutions for each.)

17.4.5 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 **do not** want to keep. Do this until only the pieces you want to keep remain highlighted, then copy the selector from the bottom pane.

Here is 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.

17.5 Scraping Presidential Statements

To demonstrate webscraping in R, we are going to collect records on presidential statements here: https://www.presidency.ucsb.edu/

Let's say we are interested in how presidents speak about "space exploration." On the website, we punch in this search term, and we get the following 325 results.

Our goal is to scrape these records and store pertinent 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(lubridate)
```

17.5.1 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 usable.

17.5.2 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 is 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 are likely to get a lot of stuff, much of which you do not 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

Challenge 1: Find the CSS Selector.

Use Selector Gadget to find the CSS selector for the document's *speaker*.

Then, modify an argument in `html_nodes` to look for this more specific CSS selector.

```
# YOUR CODE HERE
```

17.5.3 Get Attributes and Text of Elements

Once we identify elements, we want to access information in those elements. 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-congress-re
```

```
# 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 the link "points" to. It is contained in the attribute `href`:

```
speaker_link <- html_nodes(document1, ".diet-title a") %>%
  html_attr("href")

speaker_link
#> [1] "/people/president/dwight-d-eisenhower"
```

17.5.4 Let's DO This

Believe it or not, that is all you need to scrape a website. Let's apply those skills to scrape a sample document from the UCSB website – the first item in our search results.

We will collect the document's date, speaker, title, and full text.

Think: Why are we doing through all this effort to scrape just one page?

1. Date

```
document1 <- read_html("https://www.presidency.ucsb.edu/documents/special-message-the-

date <- html_nodes(document1, ".date-display-single") %>%
  html_text() %>% # Grab element text
  mdy() # Format using lubridate

date
#> [1] "1958-04-02"
```

2. Speaker

```
speaker <- html_nodes(document1, ".diet-title a") %>%
  html_text()

speaker
#> [1] "Dwight D. Eisenhower"
```

3. Title

```
title <- html_nodes(document1, "h1") %>%
  html_text()

title
#> [1] "Special Message to the Congress Relative to Space Science and Exploration."
```

4. Text

```
text <- html_nodes(document1, "div.field-docs-content") %>%
          html_text()

# This is a long document, so let's just display the first 1,000 characters
text %>% str_sub(1, 1000)
#> [1] "\n    To the Congress of the United States:\nRecent developments in long-range rockets fo
```

Challenge 2: 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 those items into a list.
- Returning that list.

```
scrape_docs <- function(URL){

  # YOUR CODE HERE

}

# Uncomment to test
# scrape_doc("https://www.presidency.ucsb.edu/documents/letter-t-keith-glennan-administrator-nat
```

Chapter 18

Text Analysis

This unit focuses on computational text analysis (or "text-as-data"). We will 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 are 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, etc., 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 will be reading documents from a CSV file in which each row is a document that includes 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 <- docs.df %>%
  mutate(text = str_conv(text, "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 preprocessing, 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 algorithms. 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 applications 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 the 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
```

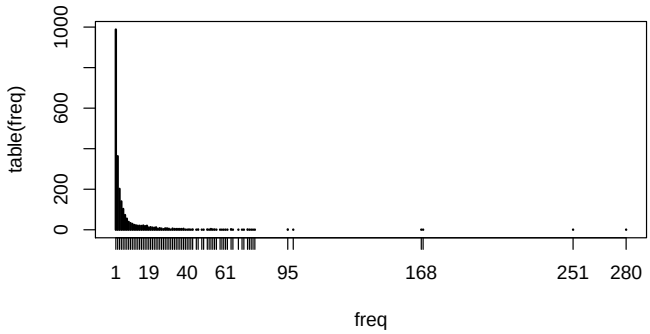
By ordering the frequencies, we can list the most frequent terms and the least frequent terms.

```
# Order
sorted <- sort(freq, decreasing = T)

# Most frequent terms
head(sorted)
#> ruler  will power   one peopl alway
#>   280   251   169   168    98    95

# Least 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?

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 50 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:

```
# Wordclouds!
set.seed(123)
wordcloud(names(sorted), sorted, max.words=100, colors=brewer.pal(6,"Dark2"))
#> Warning in wordcloud(names(sorted), sorted, max.words = 100, colors =
```

```
#> brewer.pal(6, : ruler could not be fit on page. It will not be plotted.
#> Warning in wordcloud(names(sorted), sorted, max.words = 100, colors =
#> brewer.pal(6, : power could not be fit on page. It will not be plotted.
```

Removing Sparse Terms

Somtimes we want to remove sparse terms and, thus, increase efficency. Look up the help file for the function `removeSparseTerms`. Using this function, create an object 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
#> <<DocumentTermMatrix (documents: 188, terms: 2368)>>
#> Non-/sparse entries: 11754/433430
#> Sparsity           : 97%
#> Maximal term length: 15
#> Weighting          : term frequency (tf)
dtm.s
#> <<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 dataframe in order to write it to a CSV, add metadata, etc.

First, create an object that converts the DTM to a dataframe (we first have to convert it to a matrix and then to a 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"

# Write CSV
# write.csv(dtm, "dtm.csv", row.names = F)
```

18.1.3.1 Challenge.

Using one of the datasets in the `data` directory, create a document term matrix and a wordcloud of the most common terms.

```
# YOUR CODE HERE
```

18.2 Sentiment Analysis and Dictionary Methods

To demonstrate sentiment analysis, we are 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)
```

18.2.1 Preprocessing and Setup

First, we must preprocess the corpus. Create a document-term matrix from the `lyrics` column of the `ts` dataframe. Complete the following preprocessing steps:

- Convert to lower.
- Remove stop words.
- Remove numbers.
- Remove punctuation.

Think: Why is stemming inappropriate for this application?

```
ts <- read.csv("data/taylor_swift.csv")

# Preprocess and create DTM
docs <- Corpus(VectorSource(ts$lyrics))

dtm <- DocumentTermMatrix(docs,
           control = list(tolower = TRUE,
                          removeNumbers = TRUE,
                          removePunctuation = TRUE,
                          stopwords = TRUE
                         ))

# Convert to dataframe
dtm <- as.data.frame(as.matrix(dtm))
```

Sentiment Dictionaries

We are 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 will 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 are 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 will 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 them in a dataframe
words = data.frame(word = colnames(dtm), stringsAsFactors = F)
head(words)
#>        word
#> 1      back
#> 2 backroads
#> 3       bed
#> 4   believe
#> 5   beneath
#> 6    beside

# Get their sentiment scores
words_sent <- words %>%
  left_join(sent) %>%
  mutate(score = replace_na(score, 0))
#> Joining, by = "word"
```

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!
doc_scores <- as.matrix(dtm) %*% words_sent$score

# Put the scores in the original documents dataframe
ts$sentiment <- doc_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 dataframe with words and scores) and returns a vector of sentiment scores for each text.

```
sentiment_score <- function(texts, sent_dict){
```

```
  # YOUR CODE HERE

  return(doc_scores)
}

# Uncomment to test it out!
# sentiment_score(ts$lyrics, sent_dict)
```

Challenge 2.

Using the function you wrote above, find out what the most and least positive Taylor Swift album is.

```
# YOUR CODE HERE
```

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
require(tidyverse)

# 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 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$that <- NULL # Fix weird encoding error with stop words
dtm.m$dont <- NULL

# Subset into 2 DTMs, 1 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 frequency
head(solelyobama, 10) # Get top 10 words for Obama
#> technolog      bank     innov    doesnt   teacher      loan      wont     debat
#>        31        30        30        29        26        22        22        21
#>    climat democraci
#>        19        19

solelytrump <- unlist(df[2, obama==0])
solelytrump <- solelytrump[order(solelytrump, decreasing = T)] # Order them by frequency
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
```

This is a start, but oftentimes 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 Frequencies

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 frequencies
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 frequencies, 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
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 Differences 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 metadata. One can then see how topical prevalence changes according to that metadata.

```
library(stm)
```

The data we will 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 have coded below. Notice that we are 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 the 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 are now going to estimate a topic model with 15 topics by regressing topical prevalence on region and year covariates.

Running the full model takes a **long** time to finish. For that reason, we are going to add an argument `max.em.its`, which sets the number of iterations. By keeping it low (15), we will 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,
```

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-encourag
#>       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-feder
#>       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--cent, marxist-
#>       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)
#>
#>  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 La
#>     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 workp
#>  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
```

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 Interpret 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`.

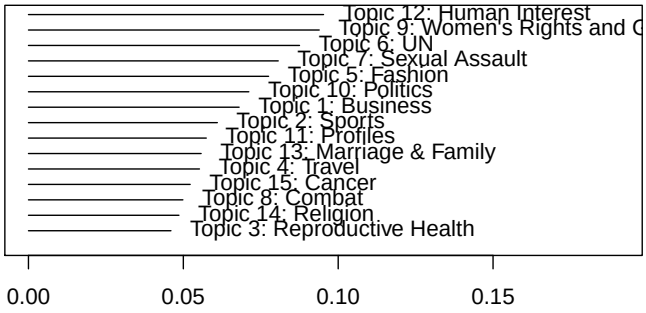
```
# Store your hand labels below
labels = c()
```

Now look at your neighbor's labels. Did you get the same results? Why or why not?

18.4.4 Analyze Topics

We are now going to see how the topics compare in terms of their prevalence across regions. 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
#> Warning: Using formula(x) is deprecated when x is a character vector of length > 1.
#>   Consider formula(paste(x, collapse = " ")) instead.

# Plot topic 9 over regions
regions = c("Asia", "EECA", "MENA", "Africa", "West", "LA")
plot.estimateEffect(prep, "REGION", method = "pointestimate", topics = 9, printlegend = TRUE, lab
```



Women's Rights