**Data Wrangling**

**with Prof. Rafael Irizarry**

**Jesse's Beast**Course Notes and Screenshots

HarvardX

**Data Wrangling**

**with Prof. Rafael Irizarry**

**Table of Contents**

[1. Section 1: Data Import 3](#_Toc25419688)

[1.1. Data Import 3](#_Toc25419689)

[1.1.1. Importing Spreadsheets 3](#_Toc25419690)

[1.1.2. Paths and the Working Directory 5](#_Toc25419691)

[1.1.3. The readr and readxl Packages 7](#_Toc25419692)

[1.1.4. Importing Data Using R-base Functions 9](#_Toc25419693)

[1.1.5. Downloading Files from the Internet 10](#_Toc25419694)

[2. Section 2: Tidy Data 12](#_Toc25419695)

[2.1. Reshaping Data 12](#_Toc25419696)

[2.1.1. Tidy Data 12](#_Toc25419697)

[2.1.2. Reshaping Data 14](#_Toc25419698)

[2.1.3. Separate and Unite 17](#_Toc25419699)

[2.2. Combining Tables 22](#_Toc25419700)

[2.2.1. Combining Tables 22](#_Toc25419701)

[2.2.2. Binding 28](#_Toc25419702)

[2.2.3. Set Operators 30](#_Toc25419703)

[2.3. Web Scraping 32](#_Toc25419704)

[2.3.1. Web Scraping 32](#_Toc25419705)

[2.3.2. CSS Selectors 35](#_Toc25419706)

[3. Section 3: String Processing 37](#_Toc25419707)

[3.1. String Processing Part 1 37](#_Toc25419708)

[3.1.1. String Parsing 37](#_Toc25419709)

[3.1.2. Defining Strings: Single and Double Quotes and How to Escape 38](#_Toc25419710)

[3.1.3. stringr Package 39](#_Toc25419711)

[3.1.4. Case Study 1: US Murders Data 40](#_Toc25419712)

[3.2. String Processing Part 2 41](#_Toc25419713)

[3.2.1. Case Study 2: Reported Heights 41](#_Toc25419714)

[3.2.2. Regex 43](#_Toc25419715)

[3.2.3. Character Classes, Anchors and Quantifiers 45](#_Toc25419716)

[3.2.4. Search and Replace with Regex 46](#_Toc25419717)

[3.2.5. Groups with Regex 49](#_Toc25419718)

[3.2.6. Testing and Improving 51](#_Toc25419719)

[3.3. String Processing Part 3 54](#_Toc25419720)

[3.3.1. Separate with Regex 54](#_Toc25419721)

[3.3.2. Using Groups and Quantifiers 55](#_Toc25419722)

[3.3.3. Putting it All Together 59](#_Toc25419723)

[3.3.4. String Splitting 60](#_Toc25419724)

[3.3.5. Case Study: Extracting a Table from a PDF 65](#_Toc25419725)

[3.3.6. Recoding 67](#_Toc25419726)

[4. Section 4: Dates, Times, and Text Mining 70](#_Toc25419727)

[4.1. Dates, Times, and Text Mining 70](#_Toc25419728)

[4.1.1. Dates and Times 70](#_Toc25419729)

[4.1.2. Text Mining 94](#_Toc25419730)

**List of Figures**

[Figure 1. Example of a csv file 3](#_Toc25419731)

[Figure 2. Look at a text file through RStudio 4](#_Toc25419732)

[Figure 3. Change working directory in R 5](#_Toc25419733)

[Figure 4. Table of available read in spreadsheets files function in R 7](#_Toc25419734)

[Figure 5. Table of functions to read in Excel data 7](#_Toc25419735)

[Figure 6. Fertility across time for Germany and South Korea 12](#_Toc25419736)

[Figure 7. Fertili ty across time for Germany and South Korea after tidying the data 15](#_Toc25419737)

[Figure 8. Diagram of use of gather and spread functions in R 16](#_Toc25419738)

[Figure 9. Population against electoral votes in 2016 24](#_Toc25419739)

[Figure 10. Diagram of different join operations in tidyverse 28](#_Toc25419740)

[Figure 11. Source code of the Wikipedia page on gun violence in the United States by state 32](#_Toc25419741)

[Figure 12. Life expectancy time series for countries in the Caribbean 68](#_Toc25419742)

[Figure 13. Life expectancy time series for countries in the Caribbean with recoded country names 69](#_Toc25419743)

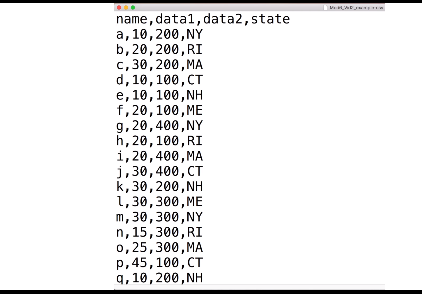
[Figure 14. Scatterplot of Ipsos poll result for Trump in 2016 71](#_Toc25419744)

1. Section 1: Data Import
   1. Data Import

### Importing Spreadsheets

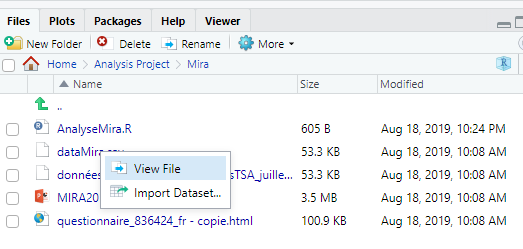
In the R basics course, we covered some of the basics of data import. We describe functions available in the default R installation. Here we present a more general discussion and introduce the tidyverse packages **readr** and **readxl.** Currently, one of the most common ways of storing and sharing data for analysis is through electronic spreadsheets. A spreadsheet file stores data in rows and columns. It is basically a file version of a data frame. When saving such tables to a computer file, one needs a way to define when a new row or column ends and the other begins. This in turn defines the cell in which single values are stored. When creating spreadsheets that are text files, like the ones you can create with a simple text editor, a new row is defined with a return and a column with some predefined special character. The most common characters to define new columns **are a comma, a semicolon, white space or a tab**

1. Example of a csv file



Apart from seeing the commas, you will also note that the first row contains column names rather than data. We call this a header. And when reading data from a spreadsheet file, it is important to know if the file has a header or not. Most reading functions assume there is header. To know if the file has a header or not, it helps to looks at the file before trying to read it in. This can be done with a text editor or with RStudio. In RStudio we can do this by navigating to the file location, clicking on that file then hitting View File

1. Look at a text file through RStudio

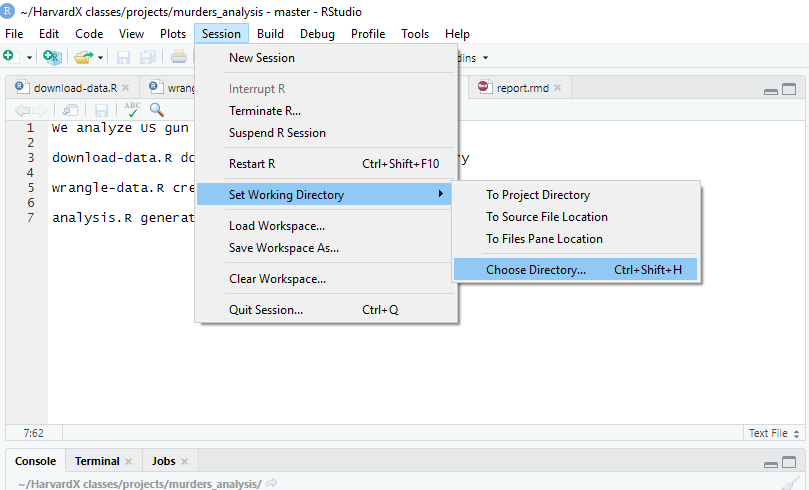


However, not all spreadsheets files are text files. Google Sheets, which are rendered on a browser, are an example. Another example is a proprietary format used by Microsoft Excel. Although it is the widely used format for data, Microsoft Excel files can’t be viewed with a text editor ( you can if you know how to read XML format 😊), if you are choosing a file format to save your own data, we recommend against using Microsoft Excel and rather choose Google Sheets as a free software tool for organizing you data if they are not sensitive.

### Paths and the Working Directory

We start by demonstrating how to read in a file that is already saved on your computer. The first step is to find the file containing your data and know its location on your file system. For this reason and others, when you are working in R, it is important to know your working directory. This is the directory in which R will save or look for files by default. You can see your working directory by typing the following command **getwed()**. You can change the working directory using the function **setwd().** If you’re using RStudio you change it by clicking on Session

1. Change working directory in R



Note that one thing that file reading functions have in common is that unless a full path is provided they search for files in the working directory. For this reason, we recommend ,for beginners, to create a directory for each analysis and keep the raw data files in that directory. To keep raw data files organized, we recommend creating a data directory inside your project directory, especially when the project involves more than one data file. Let’s look at an example. Because you may not have a data file yet, one is provided for you in the dslabs package. Once you download and install the package file will be in the external data, extdata directory that you can get by typing this command:

system.file("extdata",package="dslabs")

*"C:/Users/Jesse's Beast/Documents/R/win-library/3.6/dslabs/extdata"*

Note that the output of this function call will change depending on your operating system, how you installed R, and the version of R. But it will be consistent within your system and you’ll be able to see the files included in this directory using the function **list.files()** like this:

path<-system.file("extdata",package="dslabs")

*> list.files(path)*

*[1] "2010\_bigfive\_regents.xls"*

*[2] "carbon\_emissions.csv"*

*[3] "fertility-two-countries-example.csv"*

*[4] "HRlist2.txt"*

*[5] "life-expectancy-and-fertility-two-countries-example.csv"*

*[6] "murders.csv"*

*[7] "olive.csv"*

*[8] "RD-Mortality-Report\_2015-18-180531.pdf"*

*[9] "ssa-death-probability.csv"*

Now that we know the location of these files, we are ready to import them into R. To make the code simpler, you can move this file to your working directory. You can do this through the file system directly. But you can also do it within R itself, using the **file.copy()** function. To do this, it will help to define a variable with the full path using the **file.path()** function. Note that using paste is not recommended since Windows, Mac, Linux and Unix use different slashes for paths. The function file.path is aware of your system and chooses the correct slashes. Here’s an example:

*filename<-"murders.csv"*

*> fullpath<-file.path(path,filename)*

*> fullpath*

*[1] "C:/Users/Jesse's Beast/Documents/R/win-library/3.6/dslabs/extdata/murders.csv"*

You can now copy the file over to your working directory using the file.copy function like this:

*file.copy(fullpath,getwd())*

*[1] TRUE*

You can check if the file is now in your working directory using the file.exists function, like this:

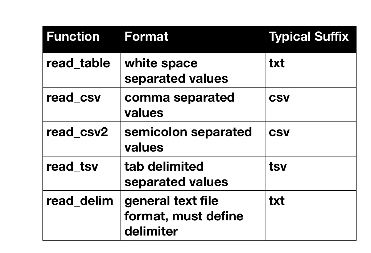
file.exists(filename)

*[1] TRUE*

### The readr and readxl Packages

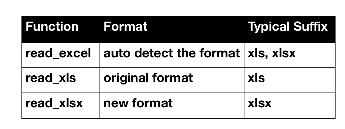
Now we are ready to read in the file, **readr()** is the tidyverse library that includes functions for reading data stored in text file spreadsheets into R. The following functions are available to read in spreadsheet files

1. Table of available read in spreadsheets files function in R



What makes these different is the type of delimiter that it works with. The **readxl** package provides function to read in data in Excel format.

1. Table of functions to read in Excel data



Two different formats and another function that tries to auto detect the format. Note that Excel permits to have more than one table in each file, they are called sheets. The function excel\_sheets give us the names of the sheets in an Excel file. These names can then be passed on the sheet argument in the three function above to read in Excel files. How do we know which of these functions to use? Note that the suffix usually tells us what type of file it is. But there’s no guarantee that these always match. To be sure, we can open the file to take a look or use functions such as **read\_lines()** that show us the first few lines of a file within R. You can do it like this:

*read\_lines("murders.csv",n\_max=3)*

*[1] "state,abb,region,population,total"*

*[2] "Alabama,AL,South,4779736,135"*

*[3] "Alaska,AK,West,710231,19"*

This also shows us if there’s a header or not. So now from this we know it’s a comma separated file and it has a header

We can know read in the data into R. We know we should use the **read\_csv()** function. Like this:

*dat<-read\_csv(filename)*

*Parsed with column specification:*

*cols(*

*state = col\_character(),*

*abb = col\_character(),*

*region = col\_character(),*

*population = col\_double(),*

*total = col\_double()*

We can also use the full path of the file like this:

*dat<-read\_csv(fullpath)*

*Parsed with column specification:*

*cols(*

*state = col\_character(),*

*abb = col\_character(),*

*region = col\_character(),*

*population = col\_double(),*

*total = col\_double()*

*)*

Note that when we run these functions, we receive a message letting us know what data types were used for each column. Also note that, the object that we just created is a tibble with the content of the file, you can see the first six lines like this:

*head(dat)*

*# A tibble: 6 x 5*

*state abb region population total*

*<chr> <chr> <chr> <dbl> <dbl>*

*1 Alabama AL South 4779736 135*

*2 Alaska AK West 710231 19*

*3 Arizona AZ West 6392017 232*

*4 Arkansas AR South 2915918 93*

*5 California CA West 37253956 1257*

*6 Colorado CO West 5029196 65*

### Importing Data Using R-base Functions

R also provides import functions we saw this earlier. These have similar names to those in the tidyverse, so don’t be confused. We have **read.table(), read.csv(), read.delim()** for example. There are a couple of important differences you should know about. To show this, we read the data using and R base function, we call this object dat2 like this:

*filename<-"murders.csv"*

*dat2<-read.csv(filename)*

One difference is that now we have a data.frame not a table. You can see it using the class function:

*class(dat2)*

*[1] "data.frame"*

The other difference is that the characters are converted to factors. Look at the class of the abbreviation column, it’s a factor:

*class(dat2$abb)*

*[1] "factor"*

*> class(dat2$region)*

*[1] "factor"*

In the original file, there were characters. This can be avoided by setting the argument stringAsFactors to FALSE:

*dat3<-read.csv(filename,stringsAsFactors = FALSE)*

*class(dat3$abb)*

*[1] "character"*

### Downloading Files from the Internet

Another common place for data to result is on the internet. When these are data files, we can download them and then import them, or we can read them indirectly from the web. For example, we know that because our DS lab package is on GitHub, the file we downloaded for the package has a URL. The read\_csv file can read these files directly, we use the URL instead of the file name when calling the function like this:

*url<-'https://raw.githubsercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv*

*dat<-read\_csv(url)*

*Parsed with column specification:*

*cols(*

*state = col\_character(),*

*abb = col\_character(),*

*region = col\_character(),*

*population = col\_double(),*

*total = col\_double()*

*)*

Now, if you want to have a local copy of the file, you can use the download.file function, like this. Two functions that are sometimes useful when downloading data from the internet are tempdir and tempfile. The first actually creates a directory with a name that is very unlikely not to be unique. Similarly, tempfile creates a character string, nor a file, that is likely to be unique file name. Look at the file name we get when we write tempfile

*tempfile()*

*[1] "C:\\Users\\JESSE'~1\\AppData\\Local\\Temp\\RtmpwnLAWs\\file144036c527aa"*

So as an example, we’ll use these commands to download a file, give it a temporary name, read it in, and then erase the files that we downloaded. We can do that using this code:

*tmp\_filename<-tempfile()*

*download.file(url,tmp\_filename)*

*dat<-read\_csv(tmp\_filename)*

*Parsed with column specification:*

*cols(*

*state = col\_character(),*

*abb = col\_character(),*

*region = col\_character(),*

*population = col\_double(),*

*total = col\_double()*

*)*

*> file.remove(tmp\_filename)*

*[1] TRUE*

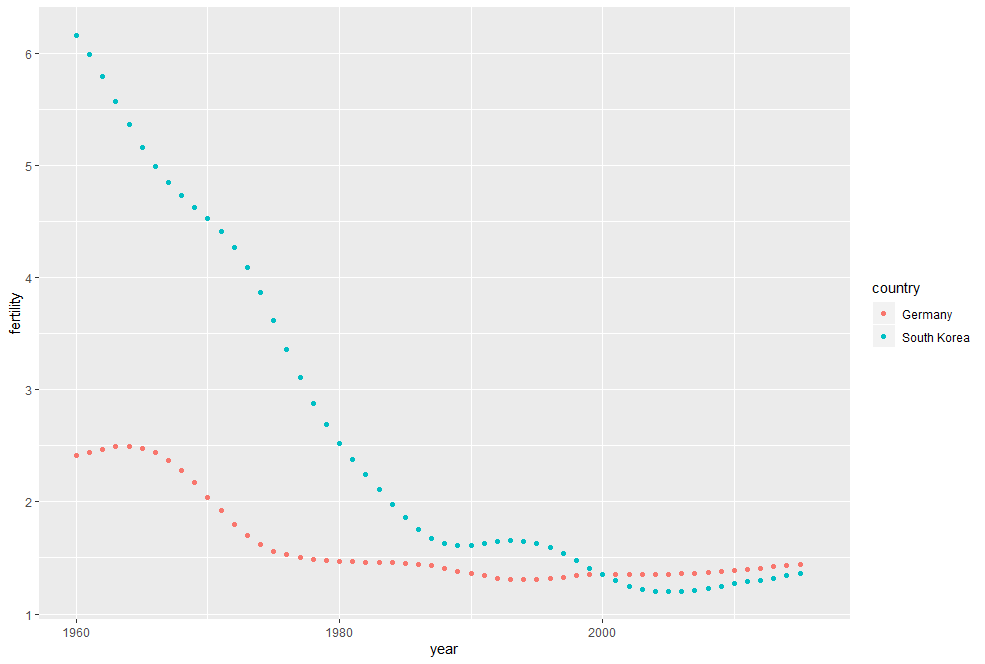
# Section 2: Tidy Data

## Reshaping Data

### Tidy Data

To help define tidy data, we go back to an example we showed in the data visualization course, in which we plotted fertility data across time for two countries South Korea and Germany, here’s the plot:

1. Fertility across time for Germany and South Korea



To make the plot, we used this subset of the data and get writing this piece of code:

*data("gapminder")*

*> tidy\_data<-gapminder%>%filter(country%in%c("South Korea","Germany"))%>%select(country,year,fertility)*

*> head(tidy\_data)*

*country year fertility*

*1 Germany 1960 2.41*

*2 South Korea 1960 6.16*

*3 Germany 1961 2.44*

*4 South Korea 1961 5.99*

*5 Germany 1962 2.47*

*6 South Korea 1962 5.79*

With the data in this format, we can quickly make the desired plot using this very simple piece of code:

*tidy\_data%>%ggplot(aes(year,fertility,color=country))+geom\_point()*

One reason this code works seamlessly is because the data is tidy. Each point in the plot is represented by a row in the table. This bring us to the definition of tidy data. Each row represents one observation and the columns represent the different variables that we have data on for those observations. If we go back to the original data provided by Gapminder, we see that it does not start out tidy. We include an example file with the data shown in this graph, mimicking the way it was originally saved in a spreadsheet. You can get to the file like this:

*path<-system.file("extdata",package="dslabs")*

*> filename<-file.path(path,"fertility-two-countries-example.csv")*

*> wide\_data<-read.csv(filename)*

After running that code, the object wide\_data includes the same information as the object tidy\_data, except it is in a different format a wide format. Gere are the first nine column of this wide data:

*select(wide\_data,country,`1960`:`1967`)*

*# A tibble: 2 x 9*

*country `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`*

*<chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*1 Germany 2.41 2.44 2.47 2.49 2.49 2.48 2.44 2.37*

*2 South Korea 6.16 5.99 5.79 5.57 5.36 5.16 4.99 4.85*

Let’s go over two important differences between the wide and tidy formats. First, in the wide format each row includes several observations. Second, one of the variables the year is stored in the header. The ggplot code we introduced earlier no longer works if we feed it the wide data. For one, there is no year available. So, to use the tidyverse we need to wrangle this data into tidy format.

### Reshaping Data

We’ve learned that having data in tidy format is what makes the tidyverse flow. After the first step in the data analysis process, importing data, a common next step is to reshape the data into a form that facilitates the rest of the analysis. The tidyr package includes several functions that are useful for tidying data. This package is included in the tidyverse. One of the most used functions in the package is gather, which converts wide data into tidy data. We’ll get to the first and second argument of gather soon, but let’s describe the third argument first. The third argument of the gather function specifies the columns that will be gathered. The default behavior for the gather function is to gather all the columns. So in most cases, we have to specify the columns.

In the example we’ve been examining, we want to gather the columns 1960,1961,… up to 2015. Those are the column names. Now let’s explain what the first argument of the gather function does. The first argument sets the name of the column that will hold the variable that are currently kept in the wide data column names. In our case, it make sense to set the name of this column to year, but we can name it anything. The second argument sets the column name for the column that will hold the values int the column cells. In this case, we’ll call it fertility since that’s the data that is in those cells. Know that nowhere in this file does it tell us that this is fertility data. We know this from the file name. This not the best way to store data but it’s the way this data was given to us.

Now the gathering code look like this:

*new\_tidy\_data<-wide\_data%>%gather(year,fertility,`1960`:`2015`)*

We can see that the data have been converted to tidy format with columns year and fertility. Look at the first six rows:

*head(new\_tidy\_data)*

*# A tibble: 6 x 3*

*country year fertility*

*<chr> <chr> <dbl>*

*1 Germany 1960 2.41*

*2 South Korea 1960 6.16*

*3 Germany 1961 2.44*

*4 South Korea 1961 5.99*

*5 Germany 1962 2.47*

*6 South Korea 1962 5.79*

Note that the only column that was not gathered was the countries column. That’s because we asked for all the other ones to be gathered. So, a somewhat quicker way to write this code is to specify which columns not to gather, rather than all the columns that will be gathered. So, the code will look simply like this:

*new\_tidy\_data<-wide\_data%>%gather(year,fertility,-country)*

The object looks a lot like the original tidy data we showed earlier. There’s just one minor difference, the data type for the year column is an integer in our original tidy data table. In our new tidy data, the one we just gathered, it’s a character. The gather function assumes that column names are characters, so we nee need a bit more wrangling begore we’re ready to make a plot. We need to convert this column to numbers. We can use as numeric if we want, but the gathered function has an argument for that. It’s the convert argument. So, the code look like this:

*new\_tidy\_data<-wide\_data%>%gather(year,fertility,-country,convert = TRUE)*

*> class(new\_tidy\_data$year)*

*[1] "integer"*

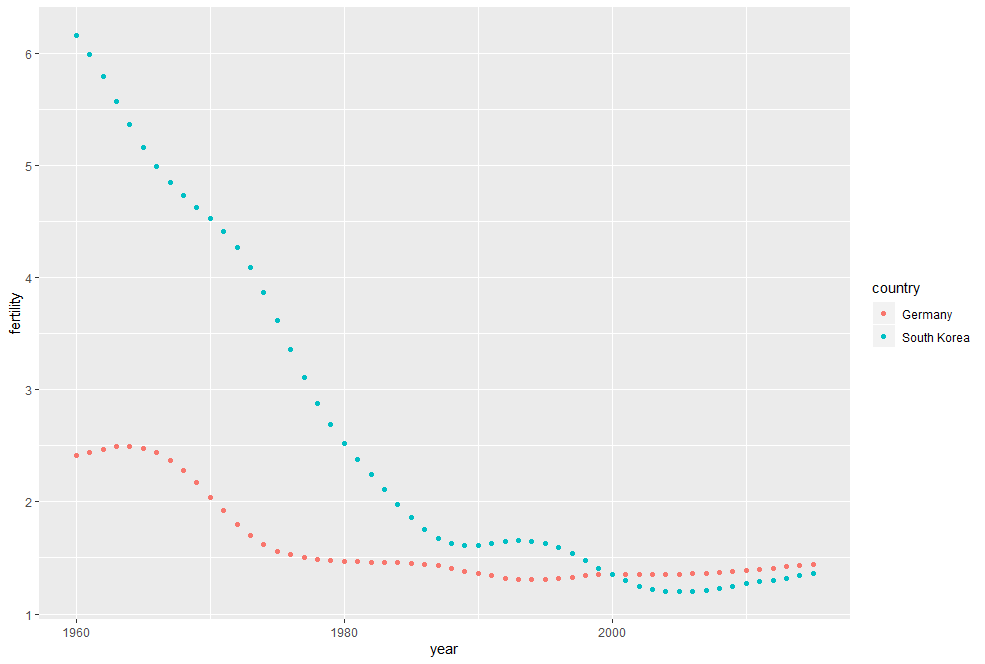
Now that the data is tidy, we can use the same ggplot commands to generate the pot we saw earlier. Like this:

*new\_tidy\_data%>%*

*+ ggplot(aes(year,fertility,color=country))+*

*+ geom\_point()*

1. Fertili ty across time for Germany and South Korea after tidying the data



Now, as we will see in later example, it is sometimes useful for data wrangling purposes to convert tidy data into the wide format data. We often use this as an intermediate step in tidying up data. The **spread()** function is basically the inverse of gather. The first argument tells spread which variables will be used as the column names. The second argument specifies which variables to use to fill out the cells. So, the code would look like this:

*new\_wide\_data<-new\_tidy\_data%>%spread(year,fertility)*

*> select(new\_wide\_data,country,`1960`:`1967`)*

*# A tibble: 2 x 9*

*country `1960` `1961` `1962` `1963`*

*<chr> <dbl> <dbl> <dbl> <dbl>*

*1 Germany 2.41 2.44 2.47 2.49*

*2 South ~ 6.16 5.99 5.79 5.57*

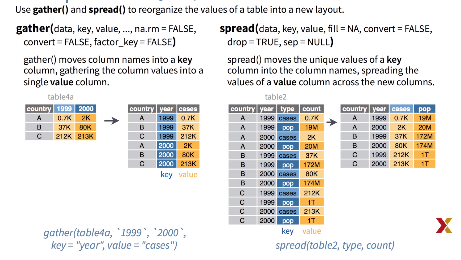
*# ... with 4 more variables: `1964` <dbl>,*

*# `1965` <dbl>, `1966` <dbl>,*

*# `1967` <dbl>*

This convert tidy data back into the wide format as you can see here. This diagram can help you remember, how these two functions work

1. Diagram of use of gather and spread functions in R



### Separate and Unite

So far we`ve seen simple example of data wrangling compared to what is usually required. In the example spreadsheets included in the dslabs package, there is, included, an example slightly more complicated. We’re going to use it to show more realistic example. This file includes two variables, life expectancy, as well as fertility, however the way it is stored is not tidy, and as we will explain, not optimal at all. You can read in the data using this piece of code:

*path<-system.file("extdata",package="dslabs")*

*> filename<-file.path(path, "life-expectancy-and-fertility-two-countries-example.csv")*

*> raw\_dat<-read\_csv(filename)*

*Parsed with column specification:*

*cols(*

*.default = col\_double(),*

*country = col\_character()*

*)*

*See spec(...) for full column specifications.*

*> select(raw\_dat,1:5)*

*# A tibble: 2 x 5*

*country `1960\_fertility` `1960\_life\_expe~*

*<chr> <dbl> <dbl>*

*1 Germany 2.41 69.3*

*2 South ~ 6.16 53.0*

*# ... with 2 more variables:*

*# `1961\_fertility` <dbl>,*

*# `1961\_life\_expectancy` <dbl>*

When we look at this table, we can see that it is in wide format. Also note that there are values for two variables with the column names encoding which column represents which variable. We can start the data wrangling with the gather function, but we should no longer use the column name Year for the new columns, since it also contains the variable type. We will call it key, that’s the default of this function, so we write this piece of code to gather the data:

*dat<-raw\_dat%>%gather(key,value,-country)*

*> head(dat)*

*# A tibble: 6 x 3*

*country key value*

*<chr> <chr> <dbl>*

*1 Germany 1960\_fertility 2.41*

*2 South Korea 1960\_fertility 6.16*

*3 Germany 1960\_life\_expectancy 69.3*

*4 South Korea 1960\_life\_expectancy 53.0*

*5 Germany 1961\_fertility 2.44*

*6 South Korea 1961\_fertility 5.99*

The result is not exactly what we refer to as tidy, since each observation is associated with two, not one, row. We want to have the value from the two variables, fertility and life expectancy, as two separate columns. The first challenge to achieve this is to separate the key column into the year and the variable type, note that the entries in this column separate the year from the variable name using an underscore. You can see it here:

*dat$key[1:5]*

*[1] "1960\_fertility"*

*[2] "1960\_fertility"*

*[3] "1960\_life\_expectancy"*

*[4] "1960\_life\_expectancy"*

*[5] "1961\_fertility"*

Encoding multiple variables in a column name is such a common problem that the readr package includes a function to separate these columns into two or more functions. The function is called **separate()**. Apart from the data, the separate function takes three arguments, the name of the column to be separated, the names to be used for the new columns, and the character that separates the variables. So a first attempt would be to write this piece of code:

*dat%>%separate(key,c("year","variable\_name"),”\_”)*

Now, because the underscore is the default separator, we can simply write the code like this:

*dat%>%separate(key,c("year","variable\_name"))*

*# A tibble: 224 x 4*

*country year variable\_name value*

*<chr> <chr> <chr> <dbl>*

*1 Germany 1960 fertility 2.41*

*2 South Korea 1960 fertility 6.16*

*3 Germany 1960 life 69.3*

*4 South Korea 1960 life 53.0*

*5 Germany 1961 fertility 2.44*

*6 South Korea 1961 fertility 5.99*

*7 Germany 1961 life 69.8*

*8 South Korea 1961 life 53.8*

*9 Germany 1962 fertility 2.47*

*10 South Korea 1962 fertility 5.79*

*# ... with 214 more rows*

*Warning message:*

*Expected 2 pieces. Additional pieces discarded in 112 rows [3, 4, 7, 8, 11, 12, 15, 16, 19, 20, 23, 24, 27, 28, 31, 32, 35, 36, 39, 40, ...].*

However, we run into a problem, note that we have received a warning, too many values at 112 locations, and that the life expectancy variable is truncated to just life. This is because the underscore is used to separate life and expectancy in the name, not just to separate year and the variable name. One of the solution is to add a third column to catch this and let the separate function know which column to fill in with missing values (NAs), in this case, when there is no third value. In this piece of code, we tell it to fill the column on the right:

*dat%>%separate(key,c("year","first\_variable\_name","second\_variable\_name"),fill="right")*

*# A tibble: 224 x 5*

*country year first\_variable\_name second\_variable\_name value*

*<chr> <chr> <chr> <chr> <dbl>*

*1 Germany 1960 fertility NA 2.41*

*2 South Korea 1960 fertility NA 6.16*

*3 Germany 1960 life expectancy 69.3*

*4 South Korea 1960 life expectancy 53.0*

*5 Germany 1961 fertility NA 2.44*

*6 South Korea 1961 fertility NA 5.99*

*7 Germany 1961 life expectancy 69.8*

*8 South Korea 1961 life expectancy 53.8*

*9 Germany 1962 fertility NA 2.47*

*10 South Korea 1962 fertility NA 5.79*

However, if we read the separate file, we find that a better approach is to merge the last two variables when there’s an extra separation using the argument extra, like this. However, we’re not done yet. We need to create a column for each variable. As we’ve learned, the spread function can do this. So now, to create tidy data, we’re actually using the spread function. So we write this piece of code:

*dat%>%separate(key,c("year","variable\_name"),sep="\_",extra="merge")%>%*

*+ spread(variable\_name,value)*

*# A tibble: 112 x 4*

*country year fertility life\_expectancy*

*<chr> <chr> <dbl> <dbl>*

*1 Germany 1960 2.41 69.3*

*2 Germany 1961 2.44 69.8*

*3 Germany 1962 2.47 70.0*

*4 Germany 1963 2.49 70.1*

*# ... with 102 more rows*

When we run it, we get a fertility and a life expectancy column. Now, it is also sometimes useful to do the inverse of separate, which is to unite two columns into one. So, although this is not optimal approach, we could have done the following to achieve the same result. We use separate like this, and then we use this code:

*dat%>%*

*+ separate(key,c("year","first\_variable\_name","second\_variable\_name"),fill="right")*

*# A tibble: 224 x 5*

*country year first\_variable\_name second\_variable\_name value*

*<chr> <chr> <chr> <chr> <dbl>*

*1 Germany 1960 fertility NA 2.41*

*2 South Korea 1960 fertility NA 6.16*

*3 Germany 1960 life expectancy 69.3*

*4 South Korea 1960 life expectancy 53.0*

*5 Germany 1961 fertility NA 2.44*

*6 South Korea 1961 fertility NA 5.99*

*7 Germany 1961 life expectancy 69.8*

*8 South Korea 1961 life expectancy 53.8*

*9 Germany 1962 fertility NA 2.47*

*10 South Korea 1962 fertility NA 5.79*

*# ... with 214 more rows*

That unites the two columns into one, and then we spread the columns with this code:

dat%>%

*+ separate(key,c("year","first\_variable\_name","second\_variable\_name"),fill="right")%>%*

*+ unite(variable\_name,first\_variable\_name,second\_variable\_name,sep="\_")%>%*

*+ spread(variable\_name,value)%>%*

*+ rename(fertility=fertility\_NA)*

*# A tibble: 112 x 4*

*country year fertility life\_expectancy*

*<chr> <chr> <dbl> <dbl>*

*1 Germany 1960 2.41 69.3*

*2 Germany 1961 2.44 69.8*

*3 Germany 1962 2.47 70.0*

*4 Germany 1963 2.49 70.1*

*5 Germany 1964 2.49 70.7*

*6 Germany 1965 2.48 70.6*

*7 Germany 1966 2.44 70.8*

*8 Germany 1967 2.37 71.0*

*9 Germany 1968 2.28 70.6*

*10 Germany 1969 2.17 70.5*

This is clearly not as efficient, but it provides an example of where you would use the unite function.

## Combining Tables

### Combining Tables

The information we need for a given analysis may not be in just one table. For example, when forecasting elections, we use the function left underscore join to combine the information from two tables. Here, we use a simple example to illustrate the general challenge of combining tables. Suppose we want to explore the relationship between population size for US states, which we have in this table:

*head(murders)*

*state abb region population total*

*1 Alabama AL South 4779736 135*

*2 Alaska AK West 710231 19*

*3 Arizona AZ West 6392017 232*

*4 Arkansas AR South 2915918 93*

*5 California CA West 37253956 1257*

*6 Colorado CO West 5029196 65*

And electoral votes which we have in this one:

*data("polls\_us\_election\_2016")*

*head(results\_us\_election\_2016)*

*state electoral\_votes clinton*

*1 California 55 61.7*

*2 Texas 38 43.2*

*3 Florida 29 47.8*

*4 New York 29 59.0*

*5 Illinois 20 55.8*

*6 Pennsylvania 20 47.9*

*trump others*

*1 31.6 6.7*

*2 52.2 4.5*

*3 49.0 3.2*

*4 36.5 4.5*

*5 38.8 5.4*

*6 48.6 3.6*

Notice that just joining these two tables together will not work since the order of the states is not quite the same. We can see this by typing this code:

*identical(results\_us\_election\_2016$state,murders$state)*

*[1] FALSE*

We see that the column names for the state names are not the same. The join functions in the dplyr package, which are based on the SQL joins, make sure that the tables are combined so that matching rows are together. The general idea is that one needs to identify one or more columns that contain the information needed to match the two tables. Then, a new table with the combined information is returned. Note what happens if we join the two tables by state using left join:

*tab<-left\_join(murders,results\_us\_election\_2016,by="state")*

*head(tab)*

*state abb region population total electoral\_votes clinton trump others*

*1 Alabama AL South 4779736 135 9 34.4 62.1 3.6*

*2 Alaska AK West 710231 19 3 36.6 51.3 12.2*

*3 Arizona AZ West 6392017 232 11 45.1 48.7 6.2*

*4 Arkansas AR South 2915918 93 6 33.7 60.6 5.8*

*5 California CA West 37253956 1257 55 61.7 31.6 6.7*

*6 Colorado CO West 5029196 65 9 48.2 43.3 8.6*

The data has been successfully joined, and we can now make a plot to explore the relationship we’re interested in by using this simple code:

*tab%>%ggplot(aes(population/10^6,electoral\_votes,label=abb))+*

*+ geom\_point()+*

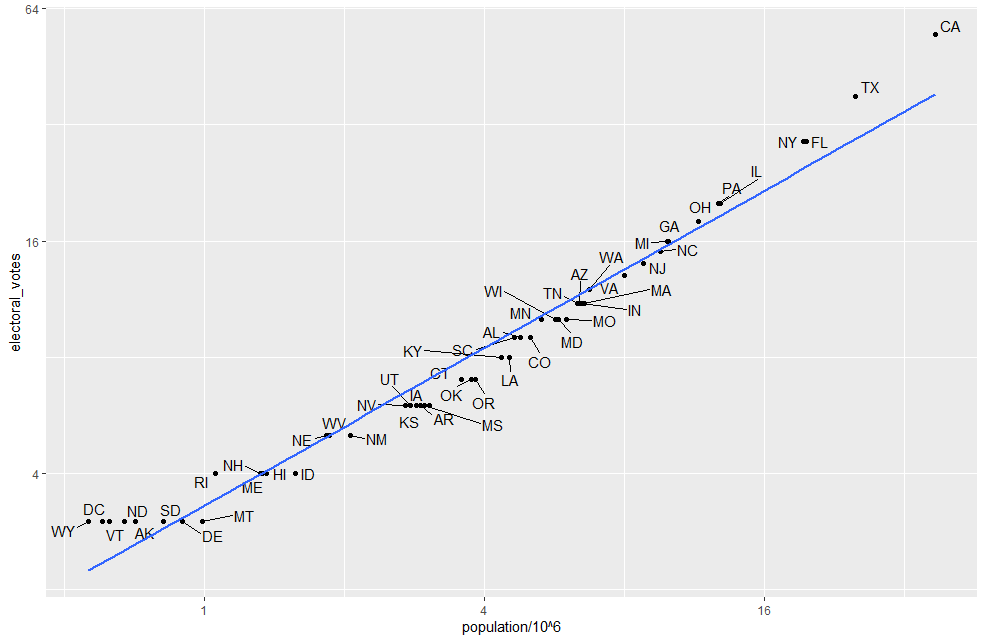
*+ geom\_text\_repel()+*

*+ scale\_x\_continuous(trans="log2")+*

*+ scale\_y\_continuous(trans="log2")+*

*+ geom\_smooth(method="lm",se=FALSE)*

1. Population against electoral votes in 2016



The plot shows that there is a relationship that’s close to linear, with about two electoral votes for every million people, but with smaller states getting a higher ratio. Now, in practice, it is not always the case that each row in one table has a matching row in the other. For this reason, we have several different ways to join. To illustrate this challenge, we’re going to take a subset of the two tables that we’ve been using. We’re going to create two objects. Tab one is going to be a subset of our first table:

*data("murders")*

*tab1<-slice(murders,1:6)%>%select(state,population)*

*tab1*

*state population*

*1 Alabama 4779736*

*2 Alaska 710231*

*3 Arizona 6392017*

*4 Arkansas 2915918*

*5 California 37253956*

*6 Colorado 5029196*

and tab two a subset of our second table:

*tab2<-slice(results\_us\_election\_2016,c(1:3,5,7:8))%>%*

*+ select(state,electoral\_votes)*

*> tab2*

*state electoral\_votes*

*1 California 55*

*2 Texas 38*

*3 Florida 29*

*4 Illinois 20*

*5 Ohio 18*

*6 Georgia 16*

However, the states contained in the two tables are going to differ. We’re going to use these two tables as examples of the different join functions. Let’s start with left join. Suppose we want a table like tab one but adding electoral votes to whatever states we have available in tab one. For this, we use left join, with tab one as the first argument, like this:

*left\_join(tab1,tab2)*

*Joining, by = "state"*

*state population electoral\_votes*

*1 Alabama 4779736 NA*

*2 Alaska 710231 NA*

*3 Arizona 6392017 NA*

*4 Arkansas 2915918 NA*

*5 California 37253956 55*

*6 Colorado 5029196 NA*

Note that NAs are added to two states in tab one that are not appearing in tab two. Also note that this function, as well as all the other joins, can receive the first argument through the pipe like this:

*tab1%>%left\_join(tab2)*

*Joining, by = "state"*

*state population electoral\_votes*

*1 Alabama 4779736 NA*

*2 Alaska 710231 NA*

*3 Arizona 6392017 NA*

*4 Arkansas 2915918 NA*

*5 California 37253956 55*

*6 Colorado 5029196 NA*

If instead of a table like tab one we want tone like tab two, we can use the right join like this:

*tab1%>%right\_join(tab2)*

*Joining, by = "state"*

*state population electoral\_votes*

*1 California 37253956 55*

*2 Texas NA 38*

*3 Florida NA 29*

*4 Illinois NA 20*

*5 Ohio NA 18*

*6 Georgia NA 16*

Notice that now the NAs are in the columns coming from tab one. Now if we want to keep only the rows that have information in both tables, we user inner join. You can think of this as an intersection, so we type:

*inner\_join(tab1,tab2)*

*Joining, by = "state"*

*state population electoral\_votes*

*1 California 37253956 55*

This one has no NAs.

Now if you want to keep all the rows and fill in the missing parts with NAs, we can use full join. You can think on this as a union. The code is simply like this:

*full\_join(tab1,tab2)*

*Joining, by = "state"*

*state population electoral\_votes*

*1 Alabama 4779736 NA*

*2 Alaska 710231 NA*

*3 Arizona 6392017 NA*

*4 Arkansas 2915918 NA*

*5 California 37253956 55*

*6 Colorado 5029196 NA*

*7 Texas NA 38*

*8 Florida NA 29*

*9 Illinois NA 20*

*10 Ohio NA 18*

*11 Georgia NA 16*

There are two more join functions we’re going to go over. They’re a little bit different because they don’t actually join the tables. Instead, they let you keep parts of one table, depending on what’s in the other. The semi join function lets us keep the part of the first table for which we have information in the second. It does not add the columns of the second. You see it by typing this:

*semi\_join(tab1,tab2)*

*Joining, by = "state"*

*state population*

*1 California 37253956*

The function anti join is the opposite of semi join. It keeps the elements of the first table for which there is no information in the second. So if we type:

*anti\_join(tab1,tab2)*

*Joining, by = "state"*

*state population*

*1 Alabama 4779736*

*2 Alaska 710231*

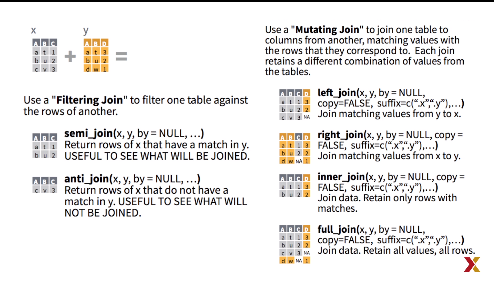
*3 Arizona 6392017*

*4 Arkansas 2915918*

*5 Colorado 5029196*

To learn more about how the joins work and to remember them you can look at this diagram:

1. Diagram of different join operations in tidyverse



### Binding

Although we have not yet used it in this series, another common way in which data sets are combined are by binding them. Regardless of the row order. Unlike the join functions, the binding functions do not try to match by a variable, but rather just combine the data sets. If the data sets don’t match by the appropriate dimension we will obtain an error. The **dplyr** function bind\_cols binds two objects by putting the columns of each together in a tibble. For example. If we quickly want to make a data frame consisting of just numbers, we can use something like this:

*bind\_cols(a=1:3,b=4:6)*

*# A tibble: 3 x 2*

*a b*

*<int> <int>*

*1 1 4*

*2 2 5*

*3 3 6*

The function requires that we assign names to the columns. Here we chose A and B, but it could be anything. Note that there’s an R-base function, cbind, that performs the same function but creates objects other than tibbles, either matrices or data frames, something else. **Bind\_cols** can also bind data frames. For example, here we break up the tab data frame and then bind them back together:

*tab1<-tab[,1:3]*

*> tab2<-tab[,4:6]*

*> tab3<-tab[,7:9]*

*> new\_tab<-bind\_cols(tab1,tab2,tab3)*

*head(new\_tab)*

*state abb region population total electoral\_votes clinton trump others*

*1 Alabama AL South 4779736 135 9 34.4 62.1 3.6*

*2 Alaska AK West 710231 19 3 36.6 51.3 12.2*

*3 Arizona AZ West 6392017 232 11 45.1 48.7 6.2*

*4 Arkansas AR South 2915918 93 6 33.7 60.6 5.8*

*5 California CA West 37253956 1257 55 61.7 31.6 6.7*

*6 Colorado CO West 5029196 65 9 48.2 43.3 8.6*

So we break it up into three with different columns, and then we simply bind them back together. So you can see that we get the original data again. The bind\_rows is similar but binds rows instead of columns. We’ll show a simple example where we take the first two rows, and the third and fourth rows, and then bind them together to get rows 1 through 4:

*tab1<-tab[1:2,]*

*tab2<-tab[3:4,]*

*bind\_rows(tab1,tab2)*

*state abb region population total electoral\_votes clinton trump others*

*1 Alabama AL South 4779736 135 9 34.4 62.1 3.6*

*2 Alaska AK West 710231 19 3 36.6 51.3 12.2*

*3 Arizona AZ West 6392017 232 11 45.1 48.7 6.2*

*4 Arkansas AR South 2915918 93 6 33.7 60.6 5.8*

This one is based on an R-base function called rbind.

### Set Operators

Another set of commands useful for combining data are the set operators. When applied to vectors, these behave as their name suggests: unions, intersect etc… and we’re going to see examples soon. However, if the tidyverse or more specifically dplyr is loaded, these functions can be used on data frames, as opposed to just on vectors. Let’s start with intersect, you can take the intersection of numeric vectors, like this:

*intersect(1:10,6:15)*

*[1] 6 7 8 9 10*

Or character vectors:

*intersect(c("a","b","c"),c("b","c","d"))*

*[1] "b" "c"*

It’s simply the intersection. But with dplyr loaded, we can also do this for tables. It’ll take the intersection of rows for tables having the same column names. So if we take the first five rows of tab and rows three through seven of tabs, and we take the intersection, it will give us rows three, four and five, which you can see here:

*tab1<-tab[1:5,]*

*tab2<-tab[3:7,]*

*intersect(tab1,tab2)*

*state abb region population total*

*1 Arizona AZ West 6392017 232*

*2 Arkansas AR South 2915918 93*

*3 California CA West 37253956 1257*

*electoral\_votes clinton trump others*

*1 11 45.1 48.7 6.2*

*2 6 33.7 60.6 5.8*

*3 55 61.7 31.6 6.7*

Similarly, union takes the union, if you apply it to vector, you get the union like this:

*union(1:10,6:15)*

*[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15*

But with dplyr loaded, we can also do this for tables having the same column names. So if we take those two tables we defined previously and take the union, we now get rows one through seven the rows that are in common, the union:

*tab1<-tab[1:2,]*

*tab2<-tab[3:4,]*

*union(tab1,tab2)*

*state abb region population total electoral\_votes clinton trump others*

*1 Alabama AL South 4779736 135 9 34.4 62.1 3.6*

*2 Alaska AK West 710231 19 3 36.6 51.3 12.2*

*3 Arizona AZ West 6392017 232 11 45.1 48.7 6.2*

*4 Arkansas AR South 2915918 93 6 33.7 60.6 5.8*

*5 California CA West 37253956 1257 55 61.7 31.6 6.7*

*6 Colorado CO West 5029196 65 9 48.2 43.3 8.6*

*7 Connecticut CT Northeast 3574097 97 7 54.6 40.9 4.5*

We can also take set differences using the function **setdiff()**. Unlike intersect and union, this function is not symmetric. For example, note that you get two different answers if you switch the arguments. You can see that using this example:

*setdiff(1:10,6:15)*

*[1] 1 2 3 4 5*

*> setdiff(6:15,1:10)*

*[1] 11 12 13 14 15*

And again, with the dplyr loaded, we can apply this to data frames. Here is an example with tab1 and tab2:

*tab1<-tab[1:5,]*

*tab2<-tab[3:7,]*

*setdiff(tab1,tab2)*

*state abb region population total electoral\_votes clinton trump others*

*1 Alabama AL South 4779736 135 9 34.4 62.1 3.6*

*2 Alaska AK West 710231 19 3 36.6 51.3 12.2*

Finally, the function setequal tell us if two sets are the same regardless of order. So for example, if I do set equals of one through five and one through six, I get false:

*setequal(1:5,1:6)*

*[1] FALSE*

Because they’re not the same vectors. But if I take set equals of one through five and five through one I get true:

*setequal(1:5,5:1)*

*[1] TRUE*

Because if you ignore the order they are the same vectors. With dplyr loaded, we can use this on data frames, as well. When applied to data frames that are not equal regardless of order, it provides a useful message letting us know how the sets differ. Look what happens if I ask if tab one and tab two are set equal:

*setequal(tab1,tab2)*

*[1] FALSE*

## Web Scraping

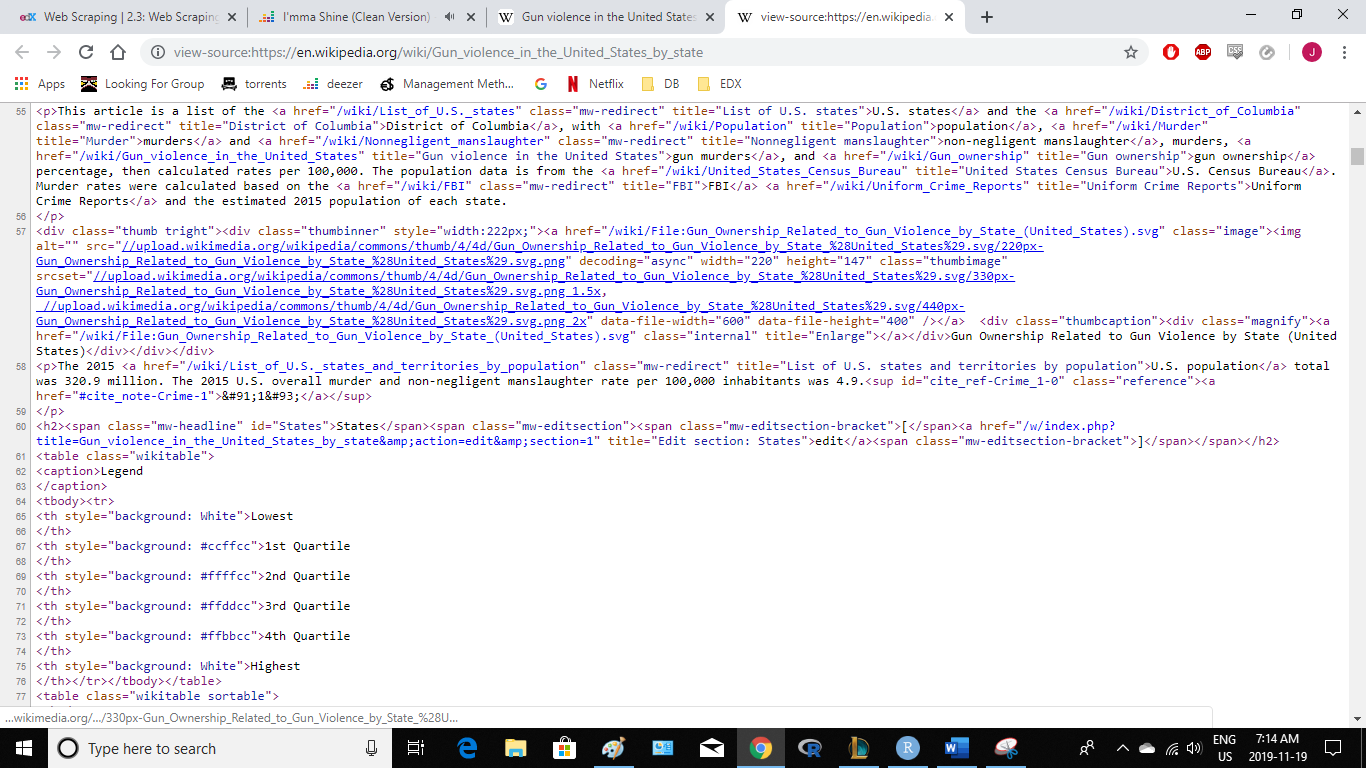
### Web Scraping

The data we need to answer questions are not always in a spreadsheet ready for us to read. For example, the US murders data set we used in R basics course originally came from this Wikipedia page:

<https://en.wikipedia.org/wiki/Gun_violence_in_the_United_States_by_state>

You can see the data table when you visit the page, but unfortunately there is no link to the data file. To make the data frame we loaded using data frames or the CSV files that was made available through Dslabs, we have to do some web scraping. Web scraping or web harvesting are the terms used to describe the process of extracting data from a website. The reason we can do this is because the information used by a browser to render web pages is received as text from a server. The text is computer code written in HyperText Markup Language or HTML. To see the code for a web page you can actually visit the page on your browser and then view the source code:

1. Source code of the Wikipedia page on gun violence in the United States by state

****

Because this code is accessible, we can download the HTML files, import it into R, and then write programs to extract the information we need from the page. However, once we look at HTML code, this might seem like a daunting task. But we will show you some convenient tools to facilitate the process. To get an idea of how web scraping works, take a look at these few lines of code from the Wikipedia page that provides the Us murders data. You can actually see the data if you look hard enough. We can also see a pattern of how it’s stored. If you know HTML, you know what these patterns are, and you can leverage this knowledge to extract what we need. We can also take advantage of a language widely used to make web pages look pretty called Cascading Style Sheets or CSS. Although we provide tools that make it possible to scrape data without knowing HTML, for data scientist, it is quite useful to learn some HTML and some CSS. Not only does this improve you scraping skills, but it might come in handy if you’re creating a web page to showcase your work. There are plenty of online courses for learning these so we don’t cover them. We’re going to provide some links in the course material. The package we’re going to learn to do web scaping is part of the tidyverse and it’s called **rvest**. The first step using this package is to import the web page into R. We can do this using this code:

*url<-"https://en.wikipedia.org/wiki/Gun\_violence\_in\_the\_United\_States\_by\_state"*

*h<-read\_html(url)*

*class(h)*

*[1] "xml\_document" "xml\_node"*

The object h stores the entire Wikipedia page. The class of this object, as you can see, is actually XML document. The rvest package is actually more general. It handles XML documents, not just HTML documents. XML is a general markup language. This language can be used to represent any kind of data. HTML is a specific type of XML, specifically developed fore representing web pages. Here we focus on HTML documents. Now, we have this object h, how do we extract a table from the object? If we print h, we don’t really see much:

*h*

*{html\_document}*

*<html class="client-nojs" lang="en" dir="ltr">*

*[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">\n<meta char ...*

*[2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject mw-editable pa ...*

Here we know that the information is stored in an HTML table. You can see this in a line of code of the HTML document we showed earlier. The different parts of HTML documents often define messages between the two symbols less than and greater than. These are referred to as nodes. The rvest package includes functions to extract nodes from HTML documents. The function html\_nodes, plural, extracts all nodes of that type, and html\_node extracts just the first node of that type. To extract the first table, we can use this very simple code:

*tab<-h%>%html\_nodes("table")*

*tab<-tab[[2]]*

Now, instead of the entire web page, we just have the HTML code for that table and we can see it by printing out tab. We are not quite there yet though, because this is clearly not a tidy data set. It’s not even a data frame. In the code we just showed, you can definitely see a pattern, and writing code to extract just the data is very doable. In fact rvest includes a function precisely for this for converting HTML tables into data frames. Here’s the code that you would use:

*tab<-tab%>%html\_table*

*class(tab)*

If you this function, you will extract the table from the HTML table and now you get a data frame. We are now much closer to having a usable data table.

Let’s change the names of the columns, which are a little bit long, and then take a look:

*tab<-tab%>%setNames(*

*c("state","population","total","murders","gun\_murders","gun\_ownership","total\_rate","murder\_rate","gun\_murder\_rate"))*

*> head(tab)*

*state population total murders gun\_murders gun\_ownership total\_rate murder\_rate*

*1 Alabama 4,853,875 348 —[a] —[a] 48.9 7.2 — [a]*

*2 Alaska 737,709 59 57 39 61.7 8.0 7.7*

*3 Arizona 6,817,565 306 278 171 32.3 4.5 4.1*

*4 Arkansas 2,977,853 181 164 110 57.9 6.1 5.5*

*5 California 38,993,940 1,861 1,861 1,275 20.1 4.8 4.8*

*6 Colorado 5,448,819 176 176 115 34.3 3.2 3.2*

*gun\_murder\_rate*

*1 — [a]*

*2 5.3*

*3 2.5*

*4 3.7*

*5 3.3*

*6 2.1*

You can see that we already have a data frame very close to what we want. However, we still have some data wrangling to do. For example, notice that some of the columns that are supposed to be numbers are actually characters, and what makes it even worse is that some of them have commas, so it makes it harder to convert to numbers. Before we continue with this, we’re going to learn a little bit more about general approaches to extracting information from websites. Then we’ll get back to our example.

### CSS Selectors

The default look of webpages made with the most basic HTML is quite unattractive. The aesthetically pleasing pages we see today are made using CSS. CSS is used to add style to webpages. The fact that all pages for a company have the same style is usually a result that they all use the same CSS file. The general way these CSS files work is by defining how each of the elements of a webpage will look. The title, headings, itemized lists, tables, and links, for example, each receive their own style including font, color, size, and distance from the margin, among others.

To do this CSS leverages patterns used to define these elements, referred to as selectors. An example of pattern we used in a previous video is table but there are many many more. If we want to grab data from a webpage and we happen to know a selector that is unique to the part of the page, we can use the html\_nodes function.

However, knowing which selector to use can be quite complicated. To demonstrate this we will try to extract the recipe name, total preparation time, and list of ingredients from this guacamole recipe. Looking at the code for this page, it seems that the task is impossibly complex. However, selector gadgets actually make this possible. SelectorGadget is piece of software that allows you to interactively determine what CSS selector you need to extract specific components from the webpage. If you plan on scraping data other than tables, we highly recommend you install it. A Chrome extension is available which permits you to turn on the gadget highlighting parts of the page as you click through, showing the necessary selector to extract those segments.

For the guacamole recipe page we already have done this and determined that we need the following selectors:

*h <- read\_html("http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609")*

*recipe <- h %>% html\_node(".o-AssetTitle\_\_a-HeadlineText") %>% html\_text()*

*prep\_time <- h %>% html\_node(".m-RecipeInfo\_\_a-Description--Total") %>% html\_text()*

*ingredients <- h %>% html\_nodes(".o-Ingredients\_\_a-Ingredient") %>% html\_text()*

You can see how complex the selectors are. In any case we are now ready to extract what we want and create a list:

*guacamole <- list(recipe, prep\_time, ingredients)*

*guacamole*

Since recipe pages from this website follow this general layout, we can use this code to create a function that extracts this information:

*get\_recipe <- function(url){*

*h <- read\_html(url)*

*recipe <- h %>% html\_node(".o-AssetTitle\_\_a-HeadlineText") %>% html\_text()*

*prep\_time <- h %>% html\_node(".m-RecipeInfo\_\_a-Description--Total") %>% html\_text()*

*ingredients <- h %>% html\_nodes(".o-Ingredients\_\_a-Ingredient") %>% html\_text()*

*return(list(recipe = recipe, prep\_time = prep\_time, ingredients = ingredients))*

*}*

and then use it on any of their webpages:

*get\_recipe("http://www.foodnetwork.com/recipes/food-network-kitchen/pancakes-recipe-1913844")*

There are several other powerful tools provided by rvest. For example, the functions html\_form, set\_values, and submit\_form permit you to query a webpage from R. This is a more advanced topic not covered here.

# Section 3: String Processing

## String Processing Part 1

### String Parsing

One of the most common data wrangling challenges involves converting or extracting numeric data contained in character strings into numeric representations required to make plots, summarize data, or fit models in R also common is processing unorganized text into meaningful variable names or categorical variables. In a web scraping section, we encounter an example related to creating the murder data set we explored in previous courses. After successfully extracting the raw data from a web page into a table, we can use code like this:

*url<-"https://en.wikipedia.org/wiki/Gun\_violence\_in\_the\_United\_States\_by\_state"*

*murders\_row<-read\_html(url)%>%html\_nodes("table")%>%html\_table()*

*murders\_row<-murders\_row[[2]]%>%setNames(c("state","population","total","murders","gun\_murders","gun\_ownership","total\_rate","murder\_rate","gun\_murder\_rate"))*

*head(murders\_row)*

*state population total murders gun\_murders gun\_ownership total\_rate murder\_rate*

*1 Alabama 4,853,875 348 —[a] —[a] 48.9 7.2 — [a]*

*2 Alaska 737,709 59 57 39 61.7 8.0 7.7*

*3 Arizona 6,817,565 306 278 171 32.3 4.5 4.1*

*4 Arkansas 2,977,853 181 164 110 57.9 6.1 5.5*

*5 California 38,993,940 1,861 1,861 1,275 20.1 4.8 4.8*

*6 Colorado 5,448,819 176 176 115 34.3 3.2 3.2*

*gun\_murder\_rate*

*1 — [a]*

*2 5.3*

*3 2.5*

*4 3.7*

*5 3.3*

*6 2.1*

We realize that two of the columns that we expected to contain numbers actually contain character, here are two examples:

*class(murders\_row$population)*

*[1] "character"*

*> class(murders\_row$total)*

*[1] "character"*

This is very common occurrence when web scraping, since web pages and other formal documents use commas in numbers to improve readability. For example, we write the number 4,853,875 like this and it’s easier to read than writing it like this with no commas. Because this is such a common task, there’s already a function, parse\_number(), that readily does this conversion. However, many of the string processing challenges a data scientist faces are unique and often unexpected. It is, therefore, quite ambitious to have a comprehensive course on these topics. So here, we use case studies that help us demonstrate how string processing is a powerful tool needed for many data wrangling challenges. Specifically, we show the original raw data used to create the data frames we have been studying in this course and describe the string processing that was needed. By going over these case studies, we’ll cover some of the most common tasks in string processing including removing unwanted characters from text, extracting numerical values from texts, finding and replacing characters, extracting specific parts of string converting free-form text to more uniform formats, and splitting strings into multiple values

### Defining Strings: Single and Double Quotes and How to Escape

To define string in R, we can use either double quotes, like this:

s<-"hello"

or single quotes like this:

s1<-'hello'

Now, what happens if your string includes double quotes? For example, if you want to write 10 inches like this: 10”. You can’t use this code:

*s<-“10””*

because this is just the string 10 followed by a double quote. If you type this into R you get an error, because you have an unclosed double quote. So to avoid this, we can use the single quotes like this:

*s<-‘10”’*

To make sure that it’s working, in R we can use the function cat. The function cat lets us see what the string looks like. So, if we type:

*cat(s)*

*10"*

We get the 10” as we wanted. Now, what if we want to use string to be 5 feet written like this 5’? In this case we use the double quotes like this:

*s<-"5'"*

you can see it works by using cat:

*cat(s)*

*5'*

So we’ve learned how to write 5 feet and 10 inches separately. What if we want to write them together to represent 5 feet and 10 inches, like this 5’10”? In this case, neither the single or double quote will work. This will give you an error, because we closed the string after the five:

*s<-‘5’10”’*

and this will give you an error, because we closed the string after the 10:

*s<-“5’10””*

Note that this doesn’t give you an error. If we type one of the above strings into R, it will get stuck waiting for you to close the open quote, and you’ll have to escape using the escape. In these situations where we can’t use either of the two quotes to write out the string we want, we need to escape the quotes, and for that we use the backslash. So we can escape either character like this:

*s<-'5\'10"'*

*cat(s)*

*5'10"*

Or we can use double quotes and escape the double quotes that represent inches, like this:

*s<-"5'10\""*

*cat(s)*

*5'10"*

Escaping characters in something we often have to use when processing strings. We will see more examples in following videos

### stringr Package

In general, string processing involves a string and a pattern. In R, we usually store strings in a character vector. In a previous video, we created an object called murders by scraping a table from the web. The population column has a character vector. The first three strings in this vector defined by the population variable can be seen here:

*murders\_row$population[1:3]*

*[1] "4,853,875" "737,709" "6,817,565"*

Note that the usual coercion to convert numbers doesn’t work here. Look what happens:

*as.numeric(murders\_row$population[1:3])*

*[1] NA NA NA*

*Warning message:*

*NAs introduced by coercion*

This is because of the commas. The string processing we want to do here is to remove the pattern comma from the string in murders\_raw$population and then coerce the numbers. In general, string processing tasks can be divided into detecting locating, extracting or replacing patterns in strings. In our example, we need to locate the comma and replace them with an empty character. Base R include function to perform all these tasks. However, they don’t follow a unifying convention, which makes it a bit hard to memorize and use. The **stringr** package basically repackages this functionality but using a more consistent approach of naming functions and ordering their arguments. For example, in stringr, all the string processing functions start with str\_, which means that if you type this and then hit Tab, R will autocomplete and show you all the available functions, which means we don’t necessarily have to memorize all the function names. Another advantage is that the string is always the first argument, which means we can move more easily using the pipe. So we’ll be focusing on the stringr package. However, because the R Base equivalent are so widely used, we’re going to show you a table that includes a map showing you the stringr function and the R Base equivalents when available.

### Case Study 1: US Murders Data

In a previous video, we scraped the web and created an object called murders\_raw. This was a table. We noted that columns needed to be parsed from characters into numbers, but that commas were making hard. We can use the **str\_detect()** function to see that the columns have commas using this code:

*commas<-function(x)any(str\_detect(x,","))*

*murders\_row%>%summarize\_all(funs(commas))*

*state population total murders gun\_murders gun\_ownership total\_rate murder\_rate*

*1 FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE*

*gun\_murder\_rate*

*1 FALSE*

We can then use the str\_replace\_all function to remove them using this code:

*test\_1<-str\_replace\_all(murders\_row$population,",","")*

*test\_1<-as.numeric(test\_1)*

We can then use the **mutate\_all()** to apply this operation to each column, since it won’t affect the columns without commas. It turns out that this operation is so common, removing commas, that readr includes the function parse\_number() specifically meant to remove non numeric characters before coercing. So we could have written it all like this:

*test\_2<-parse\_number(murders\_row$population)*

So, we can obtain our desired table using the following code. Here we’re going to use parse\_number():

*murder\_new<-murders\_row%>%mutate\_at(2:3,parse\_number)*

*murder\_new%>%head*

*state population total murders gun\_murders gun\_ownership total\_rate murder\_rate*

*1 Alabama 4853875 348 —[a] —[a] 48.9 7.2 — [a]*

*2 Alaska 737709 59 57 39 61.7 8.0 7.7*

*3 Arizona 6817565 306 278 171 32.3 4.5 4.1*

*4 Arkansas 2977853 181 164 110 57.9 6.1 5.5*

*gun\_murder\_rate*

*1 — [a]*

*2 5.3*

*3 2.5*

*4 3.7*

This case is relatively simple compared to the string processing challenges that we typically face in data science. The case studies presented in the next part are a little bit more complex.

## String Processing Part 2

### Case Study 2: Reported Heights

In previous courses, we used a heights data set in the dslabs package. The dslabs package also includes the raw data from which these heights were obtained. You can load it like this:

*library(dslabs)*

*data("reported\_heights")*

These heights were obtained using a web form in which students were asked to enter their heights into a form. They could enter anything, but the instructions asked for heights inches. We compiled over 1000 submissions, but unfortunately the column vector with the reported heights had several non-numeric entries, and as a result became a character vector. We can see it here:

*class(reported\_heights$height)*

*[1] "character"*

If we try to parse it into a number, we get a warning, There’s a lot of NAs:

*x<-as.numeric(reported\_heights$height)*

*Warning message:*

*NAs introduced by coercion*

Although most values appear to be height in inches as requested, here are the first five:

*head(x)*

*[1] 75 70 68 74 61 65*

We do end up with many NAs. You can see how many using this code:

*sum(is.na(x))*

*[1] 81*

We can see some of the entries that not successfully converted by using the filter function to keep only the entries that resulted in NAs. We can write this code:

*reported\_heights%>%mutate(new\_height=as.numeric(height))%>%*

*+ filter(is.na(new\_height))%>%*

*+ head(n=10)*

*time\_stamp sex height new\_height*

*1 2014-09-02 15:16:28 Male 5' 4" NA*

*2 2014-09-02 15:16:37 Female 165cm NA*

*3 2014-09-02 15:16:52 Male 5'7 NA*

*4 2014-09-02 15:16:56 Male >9000 NA*

*5 2014-09-02 15:16:56 Male 5'7" NA*

*6 2014-09-02 15:17:09 Female 5'3" NA*

*7 2014-09-02 15:18:00 Male 5 feet and 8.11 inches NA*

*8 2014-09-02 15:19:48 Male 5'11 NA*

*9 2014-09-04 00:46:45 Male 5'9'' NA*

*10 2014-09-04 10:29:44 Male 5'10'' NA*

*Warning message:*

*NAs introduced by coercion*

Now, look at the entries that turn out to be non-numeric. We immediately see what’s happening. Some of students did not report their heights in inches as requested. We could discard these and continue, however, many of the entries follow patterns that, in principle, we can easily convert to inches. For example, in the output above, we see various cases that use the following format x’y” with x representing feet and y representing inches. Each of these cases can be read and converted to inches by a human. For example, if you write 5’4” it is equal to 5\*12+4 which is 64 inches. So we could fix all the problematic entries by hand. However, humans are prone to making mistakes. Also, because we plan on continuing to collect data going forward, it’ll be convenient to write code that automatically does this. A first step in this type of task is to survey the problematic entries and try to define specific patterns followed by a large group of entries. The larger these groups, the more entries we can fix with a single programmatic approach. We want to find patterns that can be accurately described with a rule, such as a digit followed by a feet symbol followed by one or two digits followed by an inches symbol. To look for such patterns, it helps remove the entries that are consistent with being inches, a view only the problematic entries. We write a function to automatically do this. We only keep entries that either result in NAs when applying as numeric or are outside a range of plausible heights. We permit a range that covers about 99.99999% if the adult population. We also use **suppresWarnings()** throughout the code to avoid the warning messages we know as.numeric will give us. So here is what the function looks like:

*not\_inches<-function(x,smallest=50,tallest=84){*

*+ inches<-suppressWarnings(as.numeric(x))*

*+ ind<-is.na(inches)|inches<smallest|inches>tallest*

*+ ind*

*+ }*

We apply this function, and find that there are these many entries that are problematic:

*problems<-reported\_heights%>%*

*+ filter(not\_inches(height))%>%*

*+ .$height*

*> length(problems)*

*[1] 292*

We can now view all the cases by simply printing them. We don’t do that here because there are so many of them. But after surveying them carefully, we notice three patterns that are followed by three large groups of entries.

* A pattern of the form x’y or x’ y” or x’y\” here is 10 examples:

*pattern<-"^\\d\\s\*'\\s\*\\d{1,2}\\.\*\\d\*'\*\"\*$"*

*> str\_subset(problems,pattern)%>%head(n=10)%>%cat*

*5' 4" 5'7 5'7" 5'3" 5'11 5'9'' 5'10'' 5' 10 5'5" 5'2"*

* A pattern x.y or x,y here is 10 examples:

*pattern<-"^[4-6]\\s\*[\\.|,]\\s\*([0-9]|10|11)$"*

*> str\_subset(problems,pattern)%>%head(n=10)%>%cat*

*5.3 5.5 6.5 5.8 5.6 5,3 5.9 6,8 5.5 6.2*

* Entries reported in centimeter, here is 10 examples:

*ind<-which(between(suppressWarnings(as.numeric(problems))/2.54,54,81))*

*> ind<-ind[!is.na(ind)]*

*> problems[ind]%>%head(n=10)%>%cat*

*150 175 177 178 163 175 178 165 165 180*

Once seen that these large groups follow specific patterns, we can develop a plan of attack. Keep in mind that there is rarely just one way to perform these tasks. Here, we pick one that helps us teach several useful techniques. But surely, there is a more efficient way of performing the task we’re about to show you.

### Regex

We have seen three patterns that define many problematic entries. We will convert entries fitting the first two patterns into standardized one. We’ll then leverage this standardization to extract the feet and inches and convert to inches. We will then define a procedure for identifying entries that are in centimeters and convert these to inches. After applying these steps, we will then check again to see what entries were not fixed and see if we can tweak our approach to be more comprehensive. This is very common in data science. There’s a lot of interactive approaches that are applied. At the end, we hope to have a script that makes web-based data collection methods robust to the most common user mistakes. To achieve our goal, we will use a technique that enables us to actually detect patterns, and extract these parts we want, regular expressions or **regex**. A regex is way to describe specific patterns of a character of text that can be used to determine if a given string matches the pattern. A set of rules have been defined to do this efficiently and precisely, and we’re going to show you some examples. We can learn more about these rules by reading tutorials that can be found online. There’s also a cheat sheets that are very useful. The patterns applied to the stringr function can be a regex, rather that a standard string. We’ll learn how this works through a series of examples. Technically, any string is a regex. Perhaps the simplest example is a single character. So, the comma that we used before here is the code:

*str\_detect(murders\_row$total,pattern)*

*[1] FALSE FALSE FALSE FALSE TRUE FALSE*

*[7] FALSE FALSE FALSE TRUE FALSE FALSE*

*[13] FALSE FALSE FALSE FALSE FALSE FALSE*

*[19] FALSE FALSE FALSE FALSE FALSE FALSE*

*[25] FALSE FALSE FALSE FALSE FALSE FALSE*

*[31] FALSE FALSE FALSE FALSE FALSE FALSE*

*[37] FALSE FALSE FALSE FALSE FALSE FALSE*

*[43] FALSE TRUE FALSE FALSE FALSE FALSE*

*[49] FALSE FALSE FALSE*

Is a simple example of searching with a regex. We noted that an entry included centimeters, cm. This is also a simple example of a regex. We can show all the entries that use “cm” like this:

*str\_subset(reported\_heights$height,"cm")*

*[1] "165cm" "170 cm"*

Now let’s consider a slightly more complicated example. Let’s ask which of the following strings satisfy your pattern. We’re going to define “yes as the ones that do and “no as the ones that don’t and then create one vector of strings, called s, including both:

*yes<-c("180 cm","70 inches")*

*> no<-c("180","70''")*

*> s<-c(yes,no)*

So, we’re asking which of the strings include the pattern “cm” or the pattern “inches”. We could call str\_detect twice, like this:

*str\_detect(s,"cm")|str\_detect(s,"inches")*

*[1] TRUE TRUE FALSE FALSE*

However, we don’t need to do this. The main feature that distinguishes the regex language from plain strings is that we can use special characters. These are characters that have a meaning. We start by introducing this character “|” , which means “or”. So if you want to know if either “cm” or “inches” appears in the string we can use the regex cm bar inches like this: cm|inches and obtain the correct answer. Another special character that will be useful for identifying feet and inches values is the \d which means any digit (0,1…9). The backslash is used to distinguish it from the character “d” in R, we have to escape the backslash, so we actually have to use “\\d”. Here’s an example:

*yes<-c("5","6","5'10","5 feet","4'11")*

*> no<-c("",".","Five","six")*

*> s<-c(yes,no)*

*> pattern<-"\\d"*

*> str\_detect(s,pattern)*

*[1] TRUE TRUE TRUE TRUE TRUE FALSE*

*[7] FALSE FALSE FALSE*

We see that we actually get the right answer: 5 trues followed by four falses. We take this opportunity to introduce the very useful **str\_view()** function. This is a helpful function for troubleshooting, as it show us the first match for each string. So if we type:

str\_view(s,pattern).

The **str\_view\_all()** shows us all the matches. So if we use it here, we can see that all the digits are highlighted. There are many other special characters in regex. We’ll learn some others, but you can see most or all of them in the cheat sheet.

### Character Classes, Anchors and Quantifiers

As we`ve done already in the previous part, you’ll see that we create strings to test our regex. To do this, we define that we know should match the pattern that we’re testing, and some that we know should not. We’ll call them “yes” and “no” respectively, this permits us to check for the two types of errors, failing to match and incorrectly matching.

Character classes are used to define a series of characters that can be matched. We define character classes with the square brackets. So, for example, if we want the parent to match only if we have a 5 or a 6, we can use the regex, [56]. So we test it out, you can see that we only detect 5s and 6s:

*str\_view(s,"[56]")*

Suppose we want to match values between 4 and 7. A common way to define a character class is with ranges. So for example, if we use [0-9] is equivalent to using \d. It’s all the digits. So the pattern [4-7] will match the numbers 4, 5, 6, and 7. We can see it in this example. However, it is important to know that in regex, everything is a character. There are no numbers. So 4 is the character 4, not the number 4. Note, for example, that if we type [1-20] this does not mean 1,2,3,4,5 up to 20. It means that characters 1 through 2 and then the character 0.

Note that characters do have an order and the digits do follow the numeric order. So 0 come before 1, which comes before 2, and so on. For the same reason, we can define letters as ranges. So [a-z] are all the lower case letters, like [A-Z] are all the letters that are uppercase. If you want all the letters, then we would write it like this.

What if we want the pattern to match when we have exactly one digit? This will useful in our case studies since feet are never more than one digit. So a restriction will help us. One way to do this with regex is by using anchors which let us define pattern that must start or end at specific places. The two most common are the caret ^ and the dollar sign $ which represent the beginning and end of a string respectively. So the pattern ^\\d$ is read as start of the string followed by one digit followed by the end of the string. Note how this pattern now only detect the strings with exactly one digit. We can see it in this example. Note that the 1 in the example we just show does not match because there’s a space in front, so it’s not just one digit. For the inches part we can have one or two digits. This can be specified in regex with quantifiers. This is done by following the pattern by curly brackets with the possible number of times the previous entry repeats. So, the pattern for one or two digits is like this \\d{1,2}, so this code will do what we want:

*pattern<-"^\\d{1,2}$"*

*yes<-c("1","5","9","12")*

*no<-c("123","a4","b")*

*str\_view(c(yes,no),pattern)*

It will find all the numbers that are two digits or one digit. In this case, 1,2, 3 does not match but 1,2 does. So now, to look for one feet and inches pattern, we can add the symbol for feet and the symbol for inches after the digits. With what we learned, we can now construct an example for the pattern x feet and y inches, with the x representing feet and the y inches. It’s going to look like this:

*pattern<-"^[4-7]'\\d{1,2}\"$"*

This pattern is now getting complex, but we can look at it carefully and break it down. The caret ^ means start of the string, [4-7] one digit either 4, 5, 6, or 7, the feet symbol “, \\d{1,2} means one or two digits and then \” is the inches symbol, and then we end with the dollar sign, which means the end of the string. Let’s test it out:

*yes<-c("5'7\"","6'2\"","5'12\"")*

*no<-c("6,2\"","6.2\"","I an 5'11\"","3'2\"","64")*

*str\_detect(yes,pattern)*

*TRUE TRUE TRUE*

*str\_detect(no,pattern)*

*FALSE FALSE FALSE FALSE FALSE*

For now, we’re permitting the inches to be 12 or larger. We will add a restriction later as a regex for this is a bit more complex than we’re ready to show now.

### Search and Replace with Regex

Earlier, we talked about the object problems containing the strings that do not appear to be in inches. We can see that only these many of them match the pattern we defined. To see why this is, we show examples that expose why we don’t have more matches. Here are some examples:

*sum(str\_detect(problems,pattern))*

*14*

*problems[c(2,10,11,12,15)]%>%str\_view(pattern)*

We see that only two of them match. A first problem we see immediately is that some students wrote out the words feet and inches. We can see the entries that did this with the function **string\_subset()** like this:

*str\_subset(problems,"inches")*

*[1] "5 feet and 8.11 inches"*

*[2] "Five foot eight inches"*

*[3] "5 feet 7inches"*

*[4] "5ft 9 inches"*

*[5] "5 ft 9 inches"*

*[6] "5 feet 6 inches"*

We see several examples; we also see that some entries use the single quotes twice to represent inches instead of the double quotes. We can see some examples using the string subset option. Here they are. A first thing we can do to solve this problem is to replace the different ways of representing inches and feet with a uniform symbol. We’ll use a single quote for feet, and for inches, we’ll simply not use anything. So, 5’y will mean 5 feet and y inches. Now, if we no longer use the inches symbol at the end, we can change our pattern accordingly by taking it out of the pattern. So, our pattern will be this then:

*pattern<-"^[4-7]'\\d{1,2}$"*

If we do this replacement before the matching, we get many more matches. So, we’re going to use the string replace function to replace feet, ft foot with the feet symbol, and we’re going to replace inches in two single double quotes and slash double quotes with nothing:

*problems%>%*

*+ str\_replace("feet|ft|foot","'")%>%*

*+ str\_replace("inches|in|''|\"","")%>%*

*+ str\_detect(pattern)%>%*

*+ sum*

*[1] 48*

We get many more matches; however, we still have many cases to go. Note that in the code we just showed, we leveraged the stringr consistency and use the pipe. Another problem we have are spaces. For example the pattern 5’ 4” does not match because there is a space between the ‘ and 4 which our pattern does not permit. Spaces are characters and R does not ignore them. You can write this function to see that these two strings are not the same:

*identical("Hi","Hi ")*

*FALSE*

There is a space in one and not the other. In regex, we can represent spaces, white space, as with \s. So to find patterns like 5’ and then another digit, we can change our pattern to the following:

*pattern\_2<-"^[4-7]'\\s\\d{1,2}\"$"*

*str\_subset(problems,pattern\_2)*

*[1] "5' 4\"" "5' 11\"" "5' 7\""*

We can see that we find a few examples. We don’t need more than one regex pattern, We can use quantifiers for this as well. So we want a pattern to permit spaces but no to require them. Even if there are several spaces like this, we will still want it to match. There is a quantifier exactly for this purpose. In regex, the asterisk \* means zero or more instances of the previous character. So let’s do a quick example:

*yes<-c("AB","A1B","A11B","A111B","A1111B")*

*no<-c("A2B","A21B")*

*str\_detect(yes,"A1\*B")*

*[1] TRUE TRUE TRUE TRUE TRUE*

*str\_detect(no,"A1\*B")*

*[1] FALSE FALSE*

We see that it finds all the yes and none of the no. Note that if matches the first string which has zero 1s and all the strings which have one or more 1s. So we can improve our pattern by adding the asterisk after the space character \s. Now there are two other similar quantifiers. For none or once, we use the question mark “?”, and for one or more we can use the plus sign +. You can see how they differ by testing it out with this code:

*data.frame(string=c("AB","A1B","A11B","A111B","A1111B"),*

*+ none\_or\_more=str\_detect(yes,"A1\*B"),*

*+ none\_or\_once=str\_detect(yes,"A1?B"),*

*+ once\_or\_more=str\_detect(yes,"A1+B"))*

*string none\_or\_more none\_or\_once once\_or\_more*

*1 AB TRUE TRUE FALSE*

*2 A1B TRUE TRUE TRUE*

*3 A11B TRUE FALSE TRUE*

*4 A111B TRUE FALSE TRUE*

*5 A1111B TRUE FALSE TRUE*

We will use all three in our reported height examples as you will see later. Right now, to improve our pattern, we can add the asterisks after the \s in front and after the feet symbol to permit space between the feet symbol and the numbers. Now we match and we get a few more entries here’s the example:

*pattern<-"^[4-7]\\s\*'\\s\*\\d{1,2}$"*

*problems%>%*

*+ str\_replace("feet|ft|foot","'")%>%*

*+ str\_replace("inches|in|''|\"","")%>%*

*+ str\_detect(pattern)%>%*

*+ sum*

*[1] 53*

We might be tempted to avoid doing this by removing all the spaces with the function string replace all. However, when doing such an operation, we need to make sure that it does not have some unintended effect. In our reported heights example, this will be a problem because some entries are of x y with “ “ separating the feet from the inches. If we move all spaces, we will incorrectly turn x y into xy, which implies that 6’1” person would turn into 61 inches person instead of a 73 inch person.

### Groups with Regex

The second a large group of problematic entries were of the form x.y x,y or x y . We want to change all these to our common format, x’y. The option of just search and replace is not doable here because we would change value such as 70.5 into 70’5. Our strategy will therefore be to search for a very specific pattern that assures us feet and inches are being provided. Then for those that match, replace appropriately. Groups are a powerful aspect of regex that permits the extraction of values. Groups are defined using parentheses. They don’t affect the pattern matching per se. Instead, it permits tools to identify specific parts of the patter so we can extract them. For example, we want to change height like 5.6 to five feet, six inches. To avoid changing patterns such as 70.2, we’ll require that the first digit be between 4 and 7, we can do that using the range operation and that the second be none or more digits. We can do that using \\d\* . Let’s start by defining a simple pattern that match this:

*pattern\_without\_groups<-“^[4-7],\\d\*$”*

We want to extract the digits so that we can then form the new version using a single quote. These are two groups, so we encapsulate them with parentheses like this:

*pattern\_with\_groups<-"^([4-7]),(\\d\*)$"*

Note that we encapsulate the part of the pattern that matches the parts we want to keep, the parts we want to extract. Before we continue, notice that adding groups does not affect the detections since it only signals that we want to save what is captured by the groups. We can see that by writing this code:

*pattern\_without\_groups<-"^[4-7],\\d\*$"*

*pattern\_with\_groups<-"^([4-7]),(\\d\*)$"*

*yes<-c("5,9","5,11","6,","6,1")*

*no<-c("5'9",",","2,8","6.1.1")*

*s<-c(yes,no)*

*str\_detect(s,pattern\_without\_groups)*

*[1] TRUE TRUE TRUE TRUE FALSE FALSE*

*[7] FALSE FALSE*

*str\_detect(s,pattern\_with\_groups)*

*[1] TRUE TRUE TRUE TRUE FALSE FALSE*

*[7] FALSE FALSE*

Once we define groups, we can use a function str\_match to extract the values these groups define, like this. Look what happens if we write this code:

*str\_match(s,pattern\_with\_groups)*

*[,1] [,2] [,3]*

*[1,] "5,9" "5" "9"*

*[2,] "5,11" "5" "11"*

*[3,] "6," "6" ""*

*[4,] "6,1" "6" "1"*

*[5,] NA NA NA*

*[6,] NA NA NA*

*[7,] NA NA NA*

*[8,] NA NA NA*

Note that the second and third columns contain feet and inches respectively. The first is the original pattern that was matched. If no match occurred, we see an N/A. Now we can understand the difference between the function str\_extract and str\_match. Str\_extract extracts only strings that match a pattern not the values defined by the groups. Here’s what happens with string extract:

*str\_extract(s,pattern\_with\_groups)*

*[1] "5,9" "5,11" "6," "6,1" NA*

*[6] NA NA NA*

Another powerful aspect of groups is that you can refer to the extracted value in regex when searching and replacing. The regex special character for the i-th group is \\i. So \\1 is the value extracted from the first group, and \\2 is the value from the second group and so on. So, as a simple example, note that the following code will replace a comma by a period, but only if it is between two digits. Here’s the code:

*pattern\_with\_groups<-"^([4-7]),(\\d\*)$"*

*yes<-c("5,9","5,11","6,","6,1")*

*no<-c("5'9",",","2,8","6.1.1")*

*s<-c(yes,no)*

*str\_replace(s,pattern\_with\_groups,"\\1'\\2")*

*[1] "5'9" "5'11" "6'" "6'1" "5'9"*

*[6] "," "2,8" "6.1.1"*

We can use this to convert cases in our reported heights. Now we’re ready to define a pattern that helps us convert all the x.y, x,y, and x y’s to our preferred format. We need to adapt pattern underscore with groups to be a bit more flexible and capture all these cases. The pattern looks like this:

*pattern\_with\_groups<-"^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$"*

Let’s break it down. The caret (^) mean start of the string, then [4-7] means one digit between 4 and 7, then the [\\s\*](file:///\\s*) means none or more spaces. The next pattern means the feet symbol is either comma (,), dor(.) or at least one space. Then we have none or more spaces again, then we have none or more digits, and then the end of the string. We can see that it appears to be working. Let’s try these examples. And we’ll be able to perform the search and replace like this:

*str\_subset(problems,pattern\_with\_groups)%>%*

*+ str\_replace(pattern\_with\_groups,"\\1'\\2")%>%head*

*[1] "5'3" "5'25" "5'5" "6'5" "5'8" "5'6"*

It’s almost what we want. There’s one little problem, we have one case with 25 inches. We’ll deal with this problem in a later section.

### Testing and Improving

We have developed a powerful string processing technique that can help us catch many of the problematic entries. Now, it’s time to test our approach search for further problems, and tweak our approach for possible improvements. Let’s write a function that captures all the entries that can’t be converted into numbers, remembering that some are in centimeters. We’ll deal with those later:

*not\_inches\_or\_cm<-function(x,smallest=50,tallest=84){*

*+ inches<-suppressWarnings(as.numeric(x))*

*+ ind<-!is.na(inches)&*

*+ ((inches>=smallest & inches<=tallest)|*

*+ (inches/2.54>=smallest & inches/2.54<=tallest))*

*+ !ind*

*+ }*

*problems<-reported\_heights%>%*

*+ filter(not\_inches\_or\_cm(height))%>%*

*+ .$height*

*> length(problems)*

*[1] 200*

Let’s see how many of these we can make fit our pattern after the several processing steps we have developed. Here, we leverage the pipe one of the advantages of using stringr. We use the pipe to concatenate the different replacements that we have just performed. Then we define the pattern and then, we go and try to see how many we match:

*converted<-problems%>%*

*+ str\_replace("feet|foot|ft","'")%>% #convert feet symbols to '*

*+ str\_replace("inches|in|''|\"","")%>% #remove inches symbols*

*+ str\_replace("^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$","\\1'\\2") #change format*

*pattern<-"^[4-7]\\s\*'\\s\*\\d{1,2}$"*

*> index<-str\_detect(converted,pattern)*

*> mean(index)*

*[1] 0.615*

We are matching more than half now. Let’s examine the remaining cases. Here they are:

*converted[!index]*

*[1] "6" "165cm" "511" "6" "2"*

*[6] ">9000" "5 ' and 8.11 " "11111" "6" "103.2"*

*[11] "19" "5" "300" "6'" "6"*

*[16] "Five ' eight " "7" "214" "6" "0.7"*

*[21] "6" "2'33" "612" "1,70" "87"*

*[26] "5'7.5" "5'7.5" "111" "5' 7.78" "12"*

*[31] "6" "yyy" "89" "34" "25"*

*[36] "6" "6" "22" "684" "6"*

*[41] "1" "1" "6\*12" "87" "6"*

*[46] "1.6" "120" "120" "23" "1.7"*

*[51] "6" "5" "69" "5' 9 " "5 ' 9 "*

*[56] "6" "6" "86" "708,661" "5 ' 6 "*

*[61] "6" "649,606" "10000" "1" "728,346"*

*[66] "0" "6" "6" "6" "100"*

*[71] "88" "6" "170 cm" "7,283,465" "5"*

*[76] "5" "34"*

There is a list of problems remaining:

* Many students measuring exactly 5 or 6 feet did not enter any inches

For example, 6’, and our pattern requires that inches be included

* Some students measuring exactly 5 or 6 feet entered just that number
* Some of the inches were entered with decimal points

For example, 5’7.5” our pattern only looks for two digits.

* Some entries have spaces the end

For example, “5 ’ 9 “

* Some entries are in meters and some of these are European decimals.

For example, 1,7 is 1.7 meter.

* Two students added CM and student spelled out the numbers 5 foot, 8 inches

It is not necessarily clear that it is worth writing code to handle all these cases since they might be rare enough. However, some give us an opportunity to learn some more regex techniques. So we will show you the code that you need to fix them in the course material.

## String Processing Part 3

### Separate with Regex

In the previous section, we constructed regex that lets us identify which elements of a character vector match the feet and inches pattern. However, we want to do more. We want to extract and save the feet and number value so that we can convert them to inches when appropriate. We’re going to construct a simpler case. We’re going to make it like this:

*s<-c("5'10","6'1")*

*tab<-data.frame(x=s)*

We have already learned to use the separate functions, so we can use this code to separate out the feet part and the inches part:

*tab%>%separate(x,c("feet","inches"),sep="'")*

*feet inches*

*1 5 10*

*2 6 1*

The extract function from the tidyr package let us regex groups to extract the desired values. Here’s the equivalent code using extract to the code using separate:

*tab%>%separate(x,c("feet","inches"),regex="(\\d)'(\\d{1,2})")*

*feet inches*

*1 5 10*

*2 6 1*

So why do we need the new function extract? The reason is that groups in regex give us much more flexibility. For example, if we define an example like this and we only want the numbers, separate fails:

*s<-c("5'10","6'1\"","5'8inches")*

*tab<-data.frame(x=s)*

*tab%>%separate(x,c("feet","inches"),sep="'",fill="right")*

*feet inches*

*1 5 10*

*2 6 1"*

*3 5 8inches*

Look at what happens. But we can use extract. The regex here is a bit more complicated, as we have to permit the single quote with spaces in feet. But we also do want the double quotes included in the value, so we do not include that in the group. So we can use extract to obtain the numbers that we want using this code:

*tab%>%extract(x,c("feet","inches"),regex="(\\d)'(\\d{1,2})")*

*feet inches*

*1 5 10*

*2 6 1*

*3 5 8*

We can use separate and extract in our case study, and in the class material we have the code that finishes off the problems and extracts the height in inches for the great majority of students.

### Using Groups and Quantifiers

Four clear patterns of entries have arisen along with some other minor problems:

* 1-Many students measuring exactly 5 or 6 feet did not enter any inches. For example, 6' - our pattern requires that inches be included.
* 2-Some students measuring exactly 5 or 6 feet entered just that number.
* 3-Some of the inches were entered with decimal points. For example, 5'7.5''. Our pattern only looks for two digits.
* 4-Some entries have spaces at the end, for example 5 ' 9.
* 5-Some entries are in meters and some of these use European decimals: 1.6, 1,7.
* 6-Two students added cm.
* 7-One student spelled out the numbers: Five foot eight inches.

It is not necessarily clear that it is worth writing code to handle all these cases since they might be rare enough. However, some give us an opportunity to learn some more regex techniques so we will build a fix.

Case 1

For case 1, if we add a '0 to, for example, convert all 6 to 6'0, then our pattern will match. This can be done using groups using the following code:

*yes <- c("5", "6", "5")*

*no <- c("5'", "5''", "5'4")*

*s <- c(yes, no)*

*str\_replace(s, "^([4-7])$", "\\1'0")*

The pattern says it has to start (^), be followed with a digit between 4 and 7, and then end there ($). The parenthesis defines the group that we pass as \\1 to the replace regex.

Cases 2 and 4

We can adapt this code slightly to handle case 2 as well which covers the entry 5'. Note that the 5' is left untouched by the code above. This is because the extra ' makes the pattern not match since we have to end with a 5 or 6. To handle case 2, we want to permit the 5 or 6 to be followed by no or one symbol for feet. So we can simply add '{0,1} after the ' to do this. We can also use the none or once special character ?. As we saw previously, this is different from \* which is none or more. We now see that this code also handles the fourth case as well:

*str\_replace(s, "^([56])'?$", "\\1'0")*

Note that here we only permit 5 and 6 but not 4 and 7. This is because heights of exactly 5 and exactly 6 feet tall are quite common, so we assume those that typed 5 or 6 really meant either 60 or 72 inches. However, heights of exactly 4 or exactly 7 feet tall are so rare that, although we accept 84 as a valid entry, we assume that a 7 was entered in error.

Case 3

We can use quantifiers to deal with case 3. These entries are not matched because the inches include decimals and our pattern does not permit this. We need allow the second group to include decimals and not just digits. This means we must permit zero or one period . followed by zero or more digits. So we will use both ? and \*. Also remember that for this particular case, the period needs to be escaped since it is a special character (it means any character except a line break).

So we can adapt our pattern, currently ^[4-7]\\s\*'\\s\*\\d{1,2}$, to permit a decimal at the end:

*pattern <- "^[4-7]\\s\*'\\s\*(\\d+\\.?\\d\*)$"*

Case 5

Case 5, meters using commas, we can approach similarly to how we converted the x.y to x'y. A difference is that we require that the first digit is 1 or 2:

*yes <- c("1,7", "1, 8", "2, " )*

*no <- c("5,8", "5,3,2", "1.7")*

*s <- c(yes, no)*

*str\_replace(s, "^([12])\\s\*,\\s\*(\\d\*)$", "\\1\\.\\2")*

We will later check if the entries are meters using their numeric values.

Trimming

In general, spaces at the start or end of the string are uninformative. These can be particularly deceptive because sometimes they can be hard to see:

*s <- "Hi "*

*cat(s)*

*identical(s, "Hi")*

This is a general enough problem that there is a function dedicated to removing them: str\_trim.

*str\_trim("5 ' 9 ")*

To upper and to lower case

One of the entries writes out numbers as words: Five foot eight inches. Although not efficient, we could add 12 extra str\_replace to convert zero to 0, one to 1, and so on. To avoid having to write two separate operations for Zero and zero, One and one, etc., we can use the str\_to\_lower function to make all words lower case first:

*s <- c("Five feet eight inches")*

*str\_to\_lower(s)*

Putting it into a function

We are now ready to define a procedure that handles converting all the problematic cases.

We can now put all this together into a function that takes a string vector and tries to convert as many strings as possible to a single format. Below is a function that puts together the previous code replacements:

*convert\_format <- function(s){*

*s %>%*

*str\_replace("feet|foot|ft", "'") %>% #convert feet symbols to '*

*str\_replace\_all("inches|in|''|\"|cm|and", "") %>% #remove inches and other symbols*

*str\_replace("^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$", "\\1'\\2") %>% #change x.y, x,y x y*

*str\_replace("^([56])'?$", "\\1'0") %>% #add 0 when to 5 or 6*

*str\_replace("^([12])\\s\*,\\s\*(\\d\*)$", "\\1\\.\\2") %>% #change european decimal*

*str\_trim() #remove extra space*

*}*

We can also write a function that converts words to numbers:

*words\_to\_numbers <- function(s){*

*str\_to\_lower(s) %>%*

*str\_replace\_all("zero", "0") %>%*

*str\_replace\_all("one", "1") %>%*

*str\_replace\_all("two", "2") %>%*

*str\_replace\_all("three", "3") %>%*

*str\_replace\_all("four", "4") %>%*

*str\_replace\_all("five", "5") %>%*

*str\_replace\_all("six", "6") %>%*

*str\_replace\_all("seven", "7") %>%*

*str\_replace\_all("eight", "8") %>%*

*str\_replace\_all("nine", "9") %>%*

*str\_replace\_all("ten", "10") %>%*

*str\_replace\_all("eleven", "11")*

*}*

Now we can see which problematic entries remain:

*converted <- problems %>% words\_to\_numbers %>% convert\_format*

*remaining\_problems <- converted[not\_inches\_or\_cm(converted)]*

*pattern <- "^[4-7]\\s\*'\\s\*\\d+\\.?\\d\*$"*

*index <- str\_detect(remaining\_problems, pattern)*

*remaining\_problems[!index]*

### Putting it All Together

We are now ready to put everything we've done so far together and wrangle our reported heights data as we try to recover as many heights as possible. The code is complex but we will break it down into parts.

We start by cleaning up the height column so that the heights are closer to a feet'inches format. We added an original heights column so we can compare before and after.

Let's start by writing a function that cleans up strings so that all the feet and inches formats use the same x'y format when appropriate.

*pattern <- "^([4-7])\\s\*'\\s\*(\\d+\\.?\\d\*)$"*

*smallest <- 50*

*tallest <- 84*

*new\_heights <- reported\_heights %>%*

*mutate(original = height,*

*height = words\_to\_numbers(height) %>% convert\_format()) %>%*

*extract(height, c("feet", "inches"), regex = pattern, remove = FALSE) %>%*

*mutate\_at(c("height", "feet", "inches"), as.numeric) %>%*

*mutate(guess = 12\*feet + inches) %>%*

*mutate(height = case\_when(*

*!is.na(height) & between(height, smallest, tallest) ~ height, #inches*

*!is.na(height) & between(height/2.54, smallest, tallest) ~ height/2.54, #centimeters*

*!is.na(height) & between(height\*100/2.54, smallest, tallest) ~ height\*100/2.54, #meters*

*!is.na(guess) & inches < 12 & between(guess, smallest, tallest) ~ guess, #feet'inches*

*TRUE ~ as.numeric(NA))) %>%*

*select(-guess)*

We can check all the entries we converted using the following code:

*new\_heights %>%*

*filter(not\_inches(original)) %>%*

*select(original, height) %>%*

*arrange(height) %>%*

*View()*

Let's look at the shortest students in our dataset using the following code:

*new\_heights %>% arrange(height) %>% head(n=7)*

We see heights of 53, 54, and 55. In the original heights column, we also have 51 and 52. These short heights are very rare and it is likely that the students actually meant 5'1, 5'2, 5'3, 5'4, and 5'5. But because we are not completely sure, we will leave them as reported.

### String Splitting

Another very common data wrangling operation is string splitting. To illustrate how this comes up, we start with an illustrative example. Suppose we did not have the function read\_csv available to use. Suppose that we instead have to read a csv file using the base R function readLines like this:

*filename<-system.file("extdata/murders.csv",package="dslabs")*

*lines<-readLines(filename)*

This function reads in the data line by line to create a vector of strings:

*lines%>%head()*

*[1] "state,abb,region,population,total"*

*[2] "Alabama,AL,South,4779736,135"*

*[3] "Alaska,AK,West,710231,19"*

*[4] "Arizona,AZ,West,6392017,232"*

*[5] "Arkansas,AR,South,2915918,93"*

*[6] "California,CA,West,37253956,1257"*

In this case, one string for each row in the spreadsheet. The first six line are the following. We want to extract the values that are separated by commas for each string in the vector. The command string split does exactly this. Here is an example:

*x<-str\_split(lines,",")*

*> x%>%head()*

*[[1]]*

*[1] "state" "abb" "region"*

*[4] "population" "total"*

*[[2]]*

*[1] "Alabama" "AL" "South" "4779736"*

*[5] "135"*

*[[3]]*

*[1] "Alaska" "AK" "West" "710231"*

*[5] "19"*

*[[4]]*

*[1] "Arizona" "AZ" "West" "6392017"*

*[5] "232"*

*[[5]]*

*[1] "Arkansas" "AR" "South"*

*[4] "2915918" "93"*

*[[6]]*

*[1] "California" "CA" "West"*

*[4] "37253956" "1257"*

Note that the first entry has the column name, so we can separate that out like this.:

*col\_names<-x[[1]]*

*x<-x[-1]*

To convert our list into a data frame, we can use a shortcut provided by the map function in the **purrr** package. The **map()** function applies the same function to each element in a list. So if we want to extract the first entry of each element in x, we can write the following code using the map function:

library(purrr)

map(x,function(y) y[1])%>%head()

[[1]]

[1] "state"

[[2]]

[1] "Alabama"

[[3]]

[1] "Alaska"

[[4]]

[1] "Arizona"

[[5]]

[1] "Arkansas"

[[6]]

[1] "California"

However, because this is such a common task, purr provides a shortcut. If the second argument, instead of a function, receives an integer, it assumes that we want that entry. So the code is actually much simpler, we can write it like this:

*map(x,1)%>%head()*

*[[1]]*

*[1] "state"*

*[[2]]*

*[1] "Alabama"*

*[[3]]*

*[1] "Alaska"*

*[[4]]*

*[1] "Arizona"*

*[[5]]*

*[1] "Arkansas"*

*[[6]]*

*[1] "California"*

To force map to return a character vector instead of a list, we can use **map\_chr().** Similarly, **map\_int()** returns integers. So to create our data frame we can use the following code:

*dat <- data.frame(parse\_guess(map\_chr(x, 1)),*

*parse\_guess(map\_chr(x, 2)),*

*parse\_guess(map\_chr(x, 3)),*

*parse\_guess(map\_chr(x, 4)),*

*parse\_guess(map\_chr(x, 5))) %>%*

*setNames(col\_names)*

*dat %>% head*

*state abb region population total*

*1 Alabama AL South 4779736 135*

*2 Alaska AK West 710231 19*

*3 Arizona AZ West 6392017 232*

*4 Arkansas AR South 2915918 93*

*5 California CA West 37253956 1257*

*6 Colorado CO West 5029196 65*

Note that, using other functions included in the purrr package, we can accomplish what we just did with much more efficient code:

*dat<-x%>%*

*+ transpose()%>%*

*+ map(~parse\_guess(unlist(.)))%>%*

*+ setNames(col\_names)%>%*

*+ as.data.frame()*

It turns out that we could have avoided all this because, in the function string split, there’s an argument called simplify=true that forces the function to return a matrix instead of a list. So we could have written this:

*dat<-x%>%*

*+ transpose()%>%*

*+ map(~parse\_guess(unlist(.)))%>%*

*+ setNames(col\_names)%>%*

*+ as.data.frame()*

*> x<-str\_split(lines,",",simplify = TRUE)*

*> col\_names<-x[1,]*

*> x<-x[-1,]*

*> x%>%as\_data\_frame()%>%*

*+ setNames(col\_names)%>%*

*+ mutate\_all(parse\_guess)*

*# A tibble: 51 x 5*

*state abb region population total*

*<chr> <chr> <chr> <dbl> <dbl>*

*1 Alabama AL South 4779736 135*

*2 Alaska AK West 710231 19*

*3 Arizona AZ West 6392017 232*

*4 Arkansas AR South 2915918 93*

*5 California CA West 37253956 1257*

*6 Colorado CO West 5029196 65*

*7 Connectic~ CT Northe~ 3574097 97*

*8 Delaware DE South 897934 38*

*9 District ~ DC South 601723 99*

*10 Florida FL South 19687653 669*

*# ... with 41 more rows*

### Case Study: Extracting a Table from a PDF

One of the datasets provided in dslabs shows scientific funding rates by gender in the Netherlands:

*library(dslabs)*

*data("research\_funding\_rates")*

*research\_funding\_rates*

The data come from a paper published in the prestigious journal PNAS. However, the data are not provided in a spreadsheet; they are in a table in a PDF document. We could extract the numbers by hand, but this could lead to human error. Instead we can try to wrangle the data using R.

**Downloading the data**

We start by downloading the PDF document then importing it into R using the following code:

*library("pdftools")*

*temp\_file <- tempfile()*

*url <- "http://www.pnas.org/content/suppl/2015/09/16/1510159112.DCSupplemental/pnas.201510159SI.pdf"*

*download.file(url, temp\_file)*

*txt <- pdf\_text(temp\_file)*

*file.remove(temp\_file)*

If we examine the object text we notice that it is a character vector with an entry for each page. So we keep the page we want using the following code:

*raw\_data\_research\_funding\_rates <- txt[2]*

The steps above can be skipped because we include the raw data in the dslabs package as well:

*data("raw\_data\_research\_funding\_rates")*

Looking at the download

Examining this object:

*raw\_data\_research\_funding\_rates %>% head*

we see that it is a long string. Each line on the page, including the table rows, is separated by the symbol for newline: \n.

We can therefore create a list with the lines of the text as elements:

*tab <- str\_split(raw\_data\_research\_funding\_rates, "\n")*

Because we start off with just one element in the string, we end up with a list with just one entry:

*tab <- tab[[1]]*

By examining this object,

*tab %>% head*

we see that the information for the column names is the third and forth entires:

*the\_names\_1 <- tab[3]*

*the\_names\_2 <- tab[4]*

In the table, the column information is spread across two lines. We want to create one vector with one name for each column. We can do this using some of the functions we have just learned.

**Extracting the table data**

Let's start with the first line:

*the\_names\_1*

We want to remove the leading space and everything following the comma. We can use regex for the latter. Then we can obtain the elements by splitting using the space. We want to split only when there are 2 or more spaces to avoid splitting success rate. So we use the regex \\s{2,} as follows:

*the\_names\_1 <- the\_names\_1 %>%*

*str\_trim() %>%*

*str\_replace\_all(",\\s.", "") %>%*

*str\_split("\\s{2,}", simplify = TRUE)*

*the\_names\_1*

Now let's look at the second line:

*the\_names\_2*

Here we want to trim the leading space and then split by space as we did for the first line:

*the\_names\_2 <- the\_names\_2 %>%*

*str\_trim() %>%*

*str\_split("\\s+", simplify = TRUE)*

*the\_names\_2*

Now we can join these to generate one name for each column:

*tmp\_names <- str\_c(rep(the\_names\_1, each = 3), the\_names\_2[-1], sep = "\_")*

*the\_names <- c(the\_names\_2[1], tmp\_names) %>%*

*str\_to\_lower() %>%*

*str\_replace\_all("\\s", "\_")*

*the\_names*

Now we are ready to get the actual data. By examining the tab object, we notice that the information is in lines 6 through 14. We can use str\_split again to achieve our goal:

*new\_research\_funding\_rates <- tab[6:14] %>%*

*str\_trim %>%*

*str\_split("\\s{2,}", simplify = TRUE) %>%*

*data.frame(stringsAsFactors = FALSE) %>%*

*setNames(the\_names) %>%*

*mutate\_at(-1, parse\_number)*

*new\_research\_funding\_rates %>% head()*

We can see that the objects are identical:

*identical(research\_funding\_rates, new\_research\_funding\_rates)*

### Recoding

Another common operation involving strings is recoding the names of categorical variables. For examples, if you have a really long name for you levels, and you will be displaying them in plots, you might want to use shorter versions of the names. For example, in a character vector with country names, you might want to change United States of America to USA and United Kingdom to UK and so on. We can do this using case when. But the tidyverse offers option that are specifically designed for this task, the **recode()** function. Here’s an example showing how to rename countries with long names. We’re going to use the gapminder data set:

*library(dslabs)*

*data("gapminder")*

Suppose we want to show the life expectancy time series for countries in the Caribbean. So here’s a code that will make that plot:

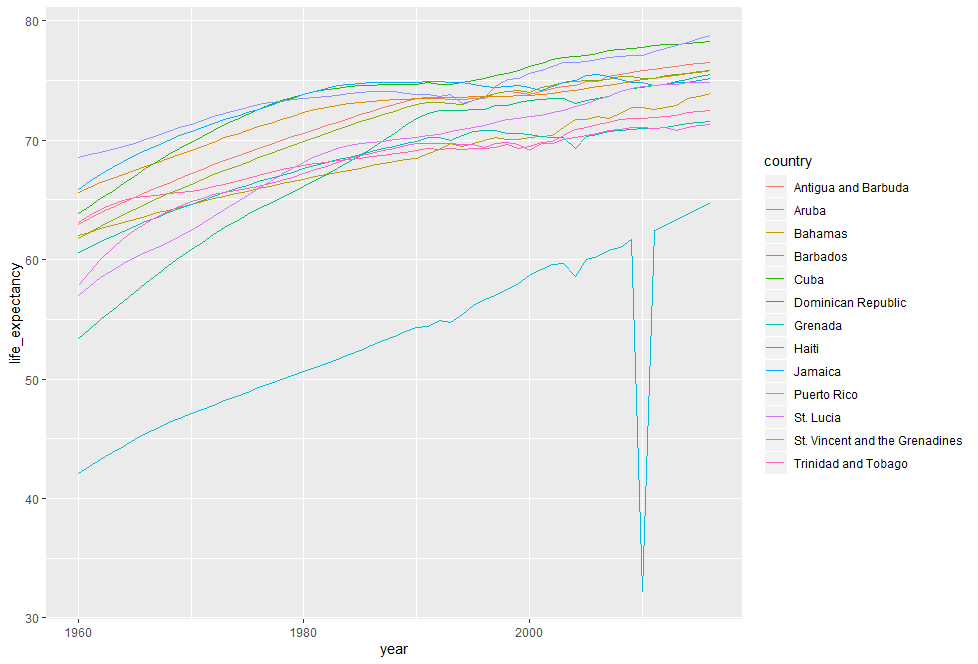
*gapminder%>%*

*+ filter(region=="Caribbean")%>%*

*+ ggplot(aes(year,life\_expectancy,color=country))+*

*+ geom\_line()*

1. Life expectancy time series for countries in the Caribbean



This is the plot we want, but much of the space is wasted to accommodate some of the long country names. Here are some of the longer ones:

*gapminder%>%*

*+ filter(region=="Caribbean")%>%*

*+ filter(str\_length(country)>=12)%>%*

*+ distinct(country)*

*country*

*1 Antigua and Barbuda*

*2 Dominican Republic*

*3 St. Vincent and the Grenadines*

*4 Trinidad and Tobago*

For example Saint Vincent and the Grenadines. We have 4 countries with names longer than 12 characters. These names appear once every year in the Gapminder data set, and once we pick nicknames, we need to change them all consistently. The recode functions can be use to do this.

Here’s an example:

*gapminder%>%filter(region=="Caribbean")%>%*

*+ mutate(country=recode(country,*

*+ `Antigua and Barbuda`="Barbuda",*

*+ `Dominican Republic`="DR",*

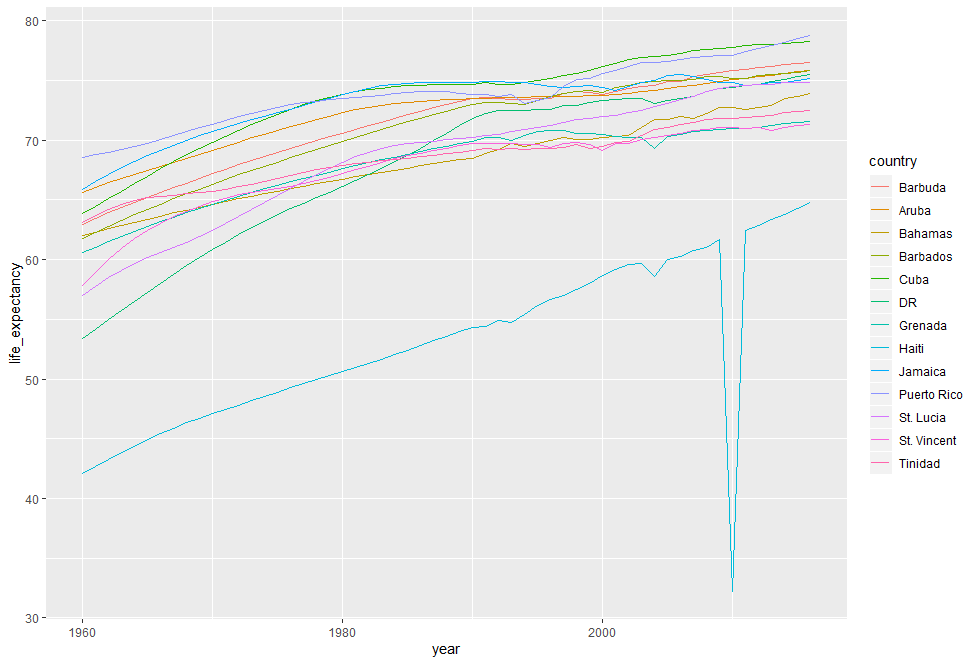
*+ `St. Vincent and the Grenadines`="St. Vincent",*

*+ `Trinidad and Tobago`="Tinidad"))%>%*

*+ ggplot(aes(year,life\_expectancy,color=country))+*

*+ geom\_line()*

1. Life expectancy time series for countries in the Caribbean with recoded country names



Notice the recode function is changing all these names to a shorter version, and it’s going to do it throughout the entire data set, as opposed to one by one. Once we do this, then we get a better-looking plot. Note that there’s other similar functions in the tidyverse. For example, **recode\_factor()** and **fct\_recoder().** These are in the forcats function in the tidyverse package

# Section 4: Dates, Times, and Text Mining

## Dates, Times, and Text Mining

### Dates and Times

Throughout this course, we have described 3 main types of vectors, numeric, character and logical. In data science projects, however, we very often encounter variables that are dates. Although we can represent dates with a string for example, November 2, 2017. Once we pick a reference day, referred to as the epoch, they can be converted to numbers. Computer languages usually use January 1, 1970 as the epoch. So November 2, 2017 is day 17204. Now, how should we represent dates and times when analyzing data in R? We could just say, use day since the epoch. But then, it’s almost impossible to interpret. If I tell you it’s November 2, 2017, you know what it means immediately. If I tell you it’s day 17204 you’ll be quite confused. Similar problems arise with times. In this case, it gets even more complicated due to time zones. For this reason, R defines a data type just for dates and times. We saw an example in polls data here it is:

*library(dslabs)*

*> data("polls\_us\_election\_2016")*

*> polls\_us\_election\_2016$startdate%>%head*

*[1] "2016-11-03" "2016-11-01" "2016-11-02"*

*[4] "2016-11-04" "2016-11-03" "2016-11-03"*

These look like strings, but if you look at their class they are not:

*class(polls\_us\_election\_2016$startdate)*

*[1] "Date"*

When you convert them to number they turn into numbers around 17000:

*as.numeric(polls\_us\_election\_2016$startdate)%>%head*

*[1] 17108 17106 17107 17109 17108 17108*

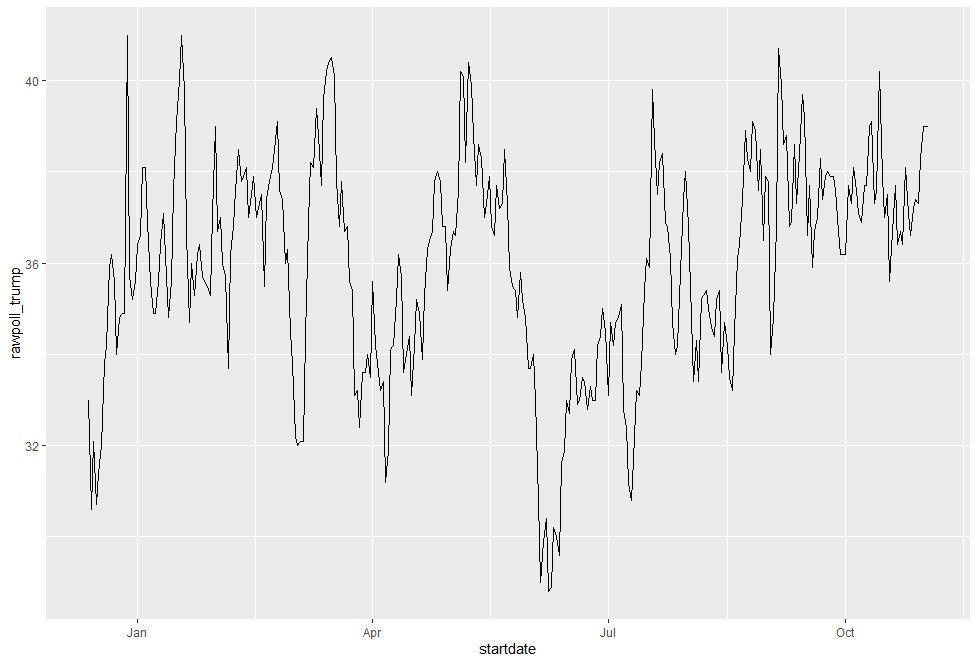
Plotting functions such as those in ggplot are aware of dates. This means that for example, a scatterplot can use a numeric representation to decide on the position of the point, but include the string and the labels, like this:

*polls\_us\_election\_2016%>%filter(pollster=="Ipsos"&state=="U.S.")%>%*

*+ ggplot(aes(startdate,rawpoll\_trump))+*

*+ geom\_line()*

1. Scatterplot of Ipsos poll result for Trump in 2016



Note that the months are displayed. Because dates are so common, the tidyverse includes functionality for dealing with dates through the **lubridate** package. We’ll take a random sample of date to show the useful things you can do with this package:

*set.seed(2)*

*dates<-sample(polls\_us\_election\_2016$startdate,10)%>%sort*

*dates*

*[1] "2016-01-19" "2016-08-06" "2016-08-26"*

*[4] "2016-09-09" "2016-09-14" "2016-09-16"*

*[7] "2016-09-29" "2016-10-04" "2016-10-12"*

*[10] "2016-10-23"*

The functions, **year()**, **month()** and **day()** extract those values, here’s an example:

*data.frame(date=days(dates),*

*+ month=month(dates),*

*+ day=day(dates),*

*+ year=year(dates))*

*date month day year*

*1 16819d 0H 0M 0S 1 19 2016*

*2 17019d 0H 0M 0S 8 6 2016*

*3 17039d 0H 0M 0S 8 26 2016*

*4 17053d 0H 0M 0S 9 9 2016*

*5 17058d 0H 0M 0S 9 14 2016*

*6 17060d 0H 0M 0S 9 16 2016*

*7 17073d 0H 0M 0S 9 29 2016*

*8 17078d 0H 0M 0S 10 4 2016*

*9 17086d 0H 0M 0S 10 12 2016*

*10 17097d 0H 0M 0S 10 23 2016*

We can also extract the month labels like this:

*month(dates,label = TRUE)*

*[1] Jan Aug Aug Sep Sep Sep Sep Oct Oct Oct*

*12 Levels: Jan < Feb < Mar < ... < Dec*

Another useful set of functions are the parsers that converts strings into dates. Here’s an example of how you can convert any kinds of strings into dates:

*x<-c(20090101,"2009-01-02","2009 01 03","2009-1-4",*

*+ "2009-1, 5","Created on 2009 1 6","200901 !!! 07")*

*> ymd(x)*

*[1] "2009-01-01" "2009-01-02" "2009-01-03"*

*[4] "2009-01-04" "2009-01-05" "2009-01-06"*

*[7] "2009-01-07"*

Another complication with dates is the fact that they can be written in different formats. The preferred format is to show year with all four digits, month with two digits and then days, or what is called ISO 8601 format. Specifically, we use the format 4 digits for Y-2 digits for M-2d digits for D. So if we order the string alphabetically it orders it by date. We saw how the function **ymd()** returns dates in this format, but what if you encountered data such as 09/01/02? This could be September 1, 2002 or January 2, 2009 or January 9, 2002. In this case, examining the entire vector of dates will help you determine what format it is by process of elimination. Once you know, you can use the many parsers provided by lubridate to convert. For example, if the string is 09/01/02, the ymd function assumes the first entry is a year, the second is a month and the third is a day. So it converts it to that format like this:

*x<-"09/01/02"*

*> ymd(x)*

*[1] "2009-01-02"*

The **mdy()** function assumes the first entry is month, then day, then year like this:

*mdy(x)*

*[1] "2002-09-01"*

Lubridate provides a function for every possibility. We list them out here:

y*dm(x)*

*[1] "2009-02-01"*

*myd(x)*

*[1] "2001-09-02"*

*dym(x)*

*[1] "2001-02-09"*

Lubridate is also useful for dealing with times. To show an example, we’re going to use the **Sys.time()** function, which, in R, get you the current time. Lubridate provides a slightly more advanced function called **now()** that permit you to define the time zone. So if I type:

*now()*

*[1] "2019-11-23 13:33:37 EST"*

If I type:

*now("GMT")*

*[1] "2019-11-23 18:34:33 GMT"*

I get the time in London.

You can see all the available time zones using the function **OlsonNames()** like this:

OlsonNames()

*[1] "Africa/Abidjan"*

*[2] "Africa/Accra"*

*[3] "Africa/Addis\_Ababa"*

*[4] "Africa/Algiers"*

*[5] "Africa/Asmara"*

*[6] "Africa/Asmera"*

*[7] "Africa/Bamako"*

*[8] "Africa/Bangui"*

*[9] "Africa/Banjul"*

*[10] "Africa/Bissau"*

*[11] "Africa/Blantyre"*

*[12] "Africa/Brazzaville"*

*[13] "Africa/Bujumbura"*

*[14] "Africa/Cairo"*

*[15] "Africa/Casablanca"*

*[16] "Africa/Ceuta"*

*[17] "Africa/Conakry"*

*[18] "Africa/Dakar"*

*[19] "Africa/Dar\_es\_Salaam"*

*[20] "Africa/Djibouti"*

*[21] "Africa/Douala"*

*[22] "Africa/El\_Aaiun"*

*[23] "Africa/Freetown"*

*[24] "Africa/Gaborone"*

*[25] "Africa/Harare"*

*[26] "Africa/Johannesburg"*

*[27] "Africa/Juba"*

*[28] "Africa/Kampala"*

*[29] "Africa/Khartoum"*

*[30] "Africa/Kigali"*

*[31] "Africa/Kinshasa"*

*[32] "Africa/Lagos"*

*[33] "Africa/Libreville"*

*[34] "Africa/Lome"*

*[35] "Africa/Luanda"*

*[36] "Africa/Lubumbashi"*

*[37] "Africa/Lusaka"*

*[38] "Africa/Malabo"*

*[39] "Africa/Maputo"*

*[40] "Africa/Maseru"*

*[41] "Africa/Mbabane"*

*[42] "Africa/Mogadishu"*

*[43] "Africa/Monrovia"*

*[44] "Africa/Nairobi"*

*[45] "Africa/Ndjamena"*

*[46] "Africa/Niamey"*

*[47] "Africa/Nouakchott"*

*[48] "Africa/Ouagadougou"*

*[49] "Africa/Porto-Novo"*

*[50] "Africa/Sao\_Tome"*

*[51] "Africa/Timbuktu"*

*[52] "Africa/Tripoli"*

*[53] "Africa/Tunis"*

*[54] "Africa/Windhoek"*

*[55] "America/Adak"*

*[56] "America/Anchorage"*

*[57] "America/Anguilla"*

*[58] "America/Antigua"*

*[59] "America/Araguaina"*

*[60] "America/Argentina/Buenos\_Aires"*

*[61] "America/Argentina/Catamarca"*

*[62] "America/Argentina/ComodRivadavia"*

*[63] "America/Argentina/Cordoba"*

*[64] "America/Argentina/Jujuy"*

*[65] "America/Argentina/La\_Rioja"*

*[66] "America/Argentina/Mendoza"*

*[67] "America/Argentina/Rio\_Gallegos"*

*[68] "America/Argentina/Salta"*

*[69] "America/Argentina/San\_Juan"*

*[70] "America/Argentina/San\_Luis"*

*[71] "America/Argentina/Tucuman"*

*[72] "America/Argentina/Ushuaia"*

*[73] "America/Aruba"*

*[74] "America/Asuncion"*

*[75] "America/Atikokan"*

*[76] "America/Atka"*

*[77] "America/Bahia"*

*[78] "America/Bahia\_Banderas"*

*[79] "America/Barbados"*

*[80] "America/Belem"*

*[81] "America/Belize"*

*[82] "America/Blanc-Sablon"*

*[83] "America/Boa\_Vista"*

*[84] "America/Bogota"*

*[85] "America/Boise"*

*[86] "America/Buenos\_Aires"*

*[87] "America/Cambridge\_Bay"*

*[88] "America/Campo\_Grande"*

*[89] "America/Cancun"*

*[90] "America/Caracas"*

*[91] "America/Catamarca"*

*[92] "America/Cayenne"*

*[93] "America/Cayman"*

*[94] "America/Chicago"*

*[95] "America/Chihuahua"*

*[96] "America/Coral\_Harbour"*

*[97] "America/Cordoba"*

*[98] "America/Costa\_Rica"*

*[99] "America/Creston"*

*[100] "America/Cuiaba"*

*[101] "America/Curacao"*

*[102] "America/Danmarkshavn"*

*[103] "America/Dawson"*

*[104] "America/Dawson\_Creek"*

*[105] "America/Denver"*

*[106] "America/Detroit"*

*[107] "America/Dominica"*

*[108] "America/Edmonton"*

*[109] "America/Eirunepe"*

*[110] "America/El\_Salvador"*

*[111] "America/Ensenada"*

*[112] "America/Fort\_Nelson"*

*[113] "America/Fort\_Wayne"*

*[114] "America/Fortaleza"*

*[115] "America/Glace\_Bay"*

*[116] "America/Godthab"*

*[117] "America/Goose\_Bay"*

*[118] "America/Grand\_Turk"*

*[119] "America/Grenada"*

*[120] "America/Guadeloupe"*

*[121] "America/Guatemala"*

*[122] "America/Guayaquil"*

*[123] "America/Guyana"*

*[124] "America/Halifax"*

*[125] "America/Havana"*

*[126] "America/Hermosillo"*

*[127] "America/Indiana/Indianapolis"*

*[128] "America/Indiana/Knox"*

*[129] "America/Indiana/Marengo"*

*[130] "America/Indiana/Petersburg"*

*[131] "America/Indiana/Tell\_City"*

*[132] "America/Indiana/Vevay"*

*[133] "America/Indiana/Vincennes"*

*[134] "America/Indiana/Winamac"*

*[135] "America/Indianapolis"*

*[136] "America/Inuvik"*

*[137] "America/Iqaluit"*

*[138] "America/Jamaica"*

*[139] "America/Jujuy"*

*[140] "America/Juneau"*

*[141] "America/Kentucky/Louisville"*

*[142] "America/Kentucky/Monticello"*

*[143] "America/Knox\_IN"*

*[144] "America/Kralendijk"*

*[145] "America/La\_Paz"*

*[146] "America/Lima"*

*[147] "America/Los\_Angeles"*

*[148] "America/Louisville"*

*[149] "America/Lower\_Princes"*

*[150] "America/Maceio"*

*[151] "America/Managua"*

*[152] "America/Manaus"*

*[153] "America/Marigot"*

*[154] "America/Martinique"*

*[155] "America/Matamoros"*

*[156] "America/Mazatlan"*

*[157] "America/Mendoza"*

*[158] "America/Menominee"*

*[159] "America/Merida"*

*[160] "America/Metlakatla"*

*[161] "America/Mexico\_City"*

*[162] "America/Miquelon"*

*[163] "America/Moncton"*

*[164] "America/Monterrey"*

*[165] "America/Montevideo"*

*[166] "America/Montreal"*

*[167] "America/Montserrat"*

*[168] "America/Nassau"*

*[169] "America/New\_York"*

*[170] "America/Nipigon"*

*[171] "America/Nome"*

*[172] "America/Noronha"*

*[173] "America/North\_Dakota/Beulah"*

*[174] "America/North\_Dakota/Center"*

*[175] "America/North\_Dakota/New\_Salem"*

*[176] "America/Ojinaga"*

*[177] "America/Panama"*

*[178] "America/Pangnirtung"*

*[179] "America/Paramaribo"*

*[180] "America/Phoenix"*

*[181] "America/Port-au-Prince"*

*[182] "America/Port\_of\_Spain"*

*[183] "America/Porto\_Acre"*

*[184] "America/Porto\_Velho"*

*[185] "America/Puerto\_Rico"*

*[186] "America/Punta\_Arenas"*

*[187] "America/Rainy\_River"*

*[188] "America/Rankin\_Inlet"*

*[189] "America/Recife"*

*[190] "America/Regina"*

*[191] "America/Resolute"*

*[192] "America/Rio\_Branco"*

*[193] "America/Rosario"*

*[194] "America/Santa\_Isabel"*

*[195] "America/Santarem"*

*[196] "America/Santiago"*

*[197] "America/Santo\_Domingo"*

*[198] "America/Sao\_Paulo"*

*[199] "America/Scoresbysund"*

*[200] "America/Shiprock"*

*[201] "America/Sitka"*

*[202] "America/St\_Barthelemy"*

*[203] "America/St\_Johns"*

*[204] "America/St\_Kitts"*

*[205] "America/St\_Lucia"*

*[206] "America/St\_Thomas"*

*[207] "America/St\_Vincent"*

*[208] "America/Swift\_Current"*

*[209] "America/Tegucigalpa"*

*[210] "America/Thule"*

*[211] "America/Thunder\_Bay"*

*[212] "America/Tijuana"*

*[213] "America/Toronto"*

*[214] "America/Tortola"*

*[215] "America/Vancouver"*

*[216] "America/Virgin"*

*[217] "America/Whitehorse"*

*[218] "America/Winnipeg"*

*[219] "America/Yakutat"*

*[220] "America/Yellowknife"*

*[221] "Antarctica/Casey"*

*[222] "Antarctica/Davis"*

*[223] "Antarctica/DumontDUrville"*

*[224] "Antarctica/Macquarie"*

*[225] "Antarctica/Mawson"*

*[226] "Antarctica/McMurdo"*

*[227] "Antarctica/Palmer"*

*[228] "Antarctica/Rothera"*

*[229] "Antarctica/South\_Pole"*

*[230] "Antarctica/Syowa"*

*[231] "Antarctica/Troll"*

*[232] "Antarctica/Vostok"*

*[233] "Arctic/Longyearbyen"*

*[234] "Asia/Aden"*

*[235] "Asia/Almaty"*

*[236] "Asia/Amman"*

*[237] "Asia/Anadyr"*

*[238] "Asia/Aqtau"*

*[239] "Asia/Aqtobe"*

*[240] "Asia/Ashgabat"*

*[241] "Asia/Ashkhabad"*

*[242] "Asia/Atyrau"*

*[243] "Asia/Baghdad"*

*[244] "Asia/Bahrain"*

*[245] "Asia/Baku"*

*[246] "Asia/Bangkok"*

*[247] "Asia/Barnaul"*

*[248] "Asia/Beirut"*

*[249] "Asia/Bishkek"*

*[250] "Asia/Brunei"*

*[251] "Asia/Calcutta"*

*[252] "Asia/Chita"*

*[253] "Asia/Choibalsan"*

*[254] "Asia/Chongqing"*

*[255] "Asia/Chungking"*

*[256] "Asia/Colombo"*

*[257] "Asia/Dacca"*

*[258] "Asia/Damascus"*

*[259] "Asia/Dhaka"*

*[260] "Asia/Dili"*

*[261] "Asia/Dubai"*

*[262] "Asia/Dushanbe"*

*[263] "Asia/Famagusta"*

*[264] "Asia/Gaza"*

*[265] "Asia/Harbin"*

*[266] "Asia/Hebron"*

*[267] "Asia/Ho\_Chi\_Minh"*

*[268] "Asia/Hong\_Kong"*

*[269] "Asia/Hovd"*

*[270] "Asia/Irkutsk"*

*[271] "Asia/Istanbul"*

*[272] "Asia/Jakarta"*

*[273] "Asia/Jayapura"*

*[274] "Asia/Jerusalem"*

*[275] "Asia/Kabul"*

*[276] "Asia/Kamchatka"*

*[277] "Asia/Karachi"*

*[278] "Asia/Kashgar"*

*[279] "Asia/Kathmandu"*

*[280] "Asia/Katmandu"*

*[281] "Asia/Khandyga"*

*[282] "Asia/Kolkata"*

*[283] "Asia/Krasnoyarsk"*

*[284] "Asia/Kuala\_Lumpur"*

*[285] "Asia/Kuching"*

*[286] "Asia/Kuwait"*

*[287] "Asia/Macao"*

*[288] "Asia/Macau"*

*[289] "Asia/Magadan"*

*[290] "Asia/Makassar"*

*[291] "Asia/Manila"*

*[292] "Asia/Muscat"*

*[293] "Asia/Nicosia"*

*[294] "Asia/Novokuznetsk"*

*[295] "Asia/Novosibirsk"*

*[296] "Asia/Omsk"*

*[297] "Asia/Oral"*

*[298] "Asia/Phnom\_Penh"*

*[299] "Asia/Pontianak"*

*[300] "Asia/Pyongyang"*

*[301] "Asia/Qatar"*

*[302] "Asia/Qostanay"*

*[303] "Asia/Qyzylorda"*

*[304] "Asia/Rangoon"*

*[305] "Asia/Riyadh"*

*[306] "Asia/Saigon"*

*[307] "Asia/Sakhalin"*

*[308] "Asia/Samarkand"*

*[309] "Asia/Seoul"*

*[310] "Asia/Shanghai"*

*[311] "Asia/Singapore"*

*[312] "Asia/Srednekolymsk"*

*[313] "Asia/Taipei"*

*[314] "Asia/Tashkent"*

*[315] "Asia/Tbilisi"*

*[316] "Asia/Tehran"*

*[317] "Asia/Tel\_Aviv"*

*[318] "Asia/Thimbu"*

*[319] "Asia/Thimphu"*

*[320] "Asia/Tokyo"*

*[321] "Asia/Tomsk"*

*[322] "Asia/Ujung\_Pandang"*

*[323] "Asia/Ulaanbaatar"*

*[324] "Asia/Ulan\_Bator"*

*[325] "Asia/Urumqi"*

*[326] "Asia/Ust-Nera"*

*[327] "Asia/Vientiane"*

*[328] "Asia/Vladivostok"*

*[329] "Asia/Yakutsk"*

*[330] "Asia/Yangon"*

*[331] "Asia/Yekaterinburg"*

*[332] "Asia/Yerevan"*

*[333] "Atlantic/Azores"*

*[334] "Atlantic/Bermuda"*

*[335] "Atlantic/Canary"*

*[336] "Atlantic/Cape\_Verde"*

*[337] "Atlantic/Faeroe"*

*[338] "Atlantic/Faroe"*

*[339] "Atlantic/Jan\_Mayen"*

*[340] "Atlantic/Madeira"*

*[341] "Atlantic/Reykjavik"*

*[342] "Atlantic/South\_Georgia"*

*[343] "Atlantic/St\_Helena"*

*[344] "Atlantic/Stanley"*

*[345] "Australia/ACT"*

*[346] "Australia/Adelaide"*

*[347] "Australia/Brisbane"*

*[348] "Australia/Broken\_Hill"*

*[349] "Australia/Canberra"*

*[350] "Australia/Currie"*

*[351] "Australia/Darwin"*

*[352] "Australia/Eucla"*

*[353] "Australia/Hobart"*

*[354] "Australia/LHI"*

*[355] "Australia/Lindeman"*

*[356] "Australia/Lord\_Howe"*

*[357] "Australia/Melbourne"*

*[358] "Australia/North"*

*[359] "Australia/NSW"*

*[360] "Australia/Perth"*

*[361] "Australia/Queensland"*

*[362] "Australia/South"*

*[363] "Australia/Sydney"*

*[364] "Australia/Tasmania"*

*[365] "Australia/Victoria"*

*[366] "Australia/West"*

*[367] "Australia/Yancowinna"*

*[368] "Brazil/Acre"*

*[369] "Brazil/DeNoronha"*

*[370] "Brazil/East"*

*[371] "Brazil/West"*

*[372] "Canada/Atlantic"*

*[373] "Canada/Central"*

*[374] "Canada/Eastern"*

*[375] "Canada/Mountain"*

*[376] "Canada/Newfoundland"*

*[377] "Canada/Pacific"*

*[378] "Canada/Saskatchewan"*

*[379] "Canada/Yukon"*

*[380] "CET"*

*[381] "Chile/Continental"*

*[382] "Chile/EasterIsland"*

*[383] "CST6CDT"*

*[384] "Cuba"*

*[385] "EET"*

*[386] "Egypt"*

*[387] "Eire"*

*[388] "EST"*

*[389] "EST5EDT"*

*[390] "Etc/GMT"*

*[391] "Etc/GMT-0"*

*[392] "Etc/GMT-1"*

*[393] "Etc/GMT-10"*

*[394] "Etc/GMT-11"*

*[395] "Etc/GMT-12"*

*[396] "Etc/GMT-13"*

*[397] "Etc/GMT-14"*

*[398] "Etc/GMT-2"*

*[399] "Etc/GMT-3"*

*[400] "Etc/GMT-4"*

*[401] "Etc/GMT-5"*

*[402] "Etc/GMT-6"*

*[403] "Etc/GMT-7"*

*[404] "Etc/GMT-8"*

*[405] "Etc/GMT-9"*

*[406] "Etc/GMT+0"*

*[407] "Etc/GMT+1"*

*[408] "Etc/GMT+10"*

*[409] "Etc/GMT+11"*

*[410] "Etc/GMT+12"*

*[411] "Etc/GMT+2"*

*[412] "Etc/GMT+3"*

*[413] "Etc/GMT+4"*

*[414] "Etc/GMT+5"*

*[415] "Etc/GMT+6"*

*[416] "Etc/GMT+7"*

*[417] "Etc/GMT+8"*

*[418] "Etc/GMT+9"*

*[419] "Etc/GMT0"*

*[420] "Etc/Greenwich"*

*[421] "Etc/UCT"*

*[422] "Etc/Universal"*

*[423] "Etc/UTC"*

*[424] "Etc/Zulu"*

*[425] "Europe/Amsterdam"*

*[426] "Europe/Andorra"*

*[427] "Europe/Astrakhan"*

*[428] "Europe/Athens"*

*[429] "Europe/Belfast"*

*[430] "Europe/Belgrade"*

*[431] "Europe/Berlin"*

*[432] "Europe/Bratislava"*

*[433] "Europe/Brussels"*

*[434] "Europe/Bucharest"*

*[435] "Europe/Budapest"*

*[436] "Europe/Busingen"*

*[437] "Europe/Chisinau"*

*[438] "Europe/Copenhagen"*

*[439] "Europe/Dublin"*

*[440] "Europe/Gibraltar"*

*[441] "Europe/Guernsey"*

*[442] "Europe/Helsinki"*

*[443] "Europe/Isle\_of\_Man"*

*[444] "Europe/Istanbul"*

*[445] "Europe/Jersey"*

*[446] "Europe/Kaliningrad"*

*[447] "Europe/Kiev"*

*[448] "Europe/Kirov"*

*[449] "Europe/Lisbon"*

*[450] "Europe/Ljubljana"*

*[451] "Europe/London"*

*[452] "Europe/Luxembourg"*

*[453] "Europe/Madrid"*

*[454] "Europe/Malta"*

*[455] "Europe/Mariehamn"*

*[456] "Europe/Minsk"*

*[457] "Europe/Monaco"*

*[458] "Europe/Moscow"*

*[459] "Europe/Nicosia"*

*[460] "Europe/Oslo"*

*[461] "Europe/Paris"*

*[462] "Europe/Podgorica"*

*[463] "Europe/Prague"*

*[464] "Europe/Riga"*

*[465] "Europe/Rome"*

*[466] "Europe/Samara"*

*[467] "Europe/San\_Marino"*

*[468] "Europe/Sarajevo"*

*[469] "Europe/Saratov"*

*[470] "Europe/Simferopol"*

*[471] "Europe/Skopje"*

*[472] "Europe/Sofia"*

*[473] "Europe/Stockholm"*

*[474] "Europe/Tallinn"*

*[475] "Europe/Tirane"*

*[476] "Europe/Tiraspol"*

*[477] "Europe/Ulyanovsk"*

*[478] "Europe/Uzhgorod"*

*[479] "Europe/Vaduz"*

*[480] "Europe/Vatican"*

*[481] "Europe/Vienna"*

*[482] "Europe/Vilnius"*

*[483] "Europe/Volgograd"*

*[484] "Europe/Warsaw"*

*[485] "Europe/Zagreb"*

*[486] "Europe/Zaporozhye"*

*[487] "Europe/Zurich"*

*[488] "GB"*

*[489] "GB-Eire"*

*[490] "GMT"*

*[491] "GMT-0"*

*[492] "GMT+0"*

*[493] "GMT0"*

*[494] "Greenwich"*

*[495] "Hongkong"*

*[496] "HST"*

*[497] "Iceland"*

*[498] "Indian/Antananarivo"*

*[499] "Indian/Chagos"*

*[500] "Indian/Christmas"*

*[501] "Indian/Cocos"*

*[502] "Indian/Comoro"*

*[503] "Indian/Kerguelen"*

*[504] "Indian/Mahe"*

*[505] "Indian/Maldives"*

*[506] "Indian/Mauritius"*

*[507] "Indian/Mayotte"*

*[508] "Indian/Reunion"*

*[509] "Iran"*

*[510] "Israel"*

*[511] "Jamaica"*

*[512] "Japan"*

*[513] "Kwajalein"*

*[514] "Libya"*

*[515] "MET"*

*[516] "Mexico/BajaNorte"*

*[517] "Mexico/BajaSur"*

*[518] "Mexico/General"*

*[519] "MST"*

*[520] "MST7MDT"*

*[521] "Navajo"*

*[522] "NZ"*

*[523] "NZ-CHAT"*

*[524] "Pacific/Apia"*

*[525] "Pacific/Auckland"*

*[526] "Pacific/Bougainville"*

*[527] "Pacific/Chatham"*

*[528] "Pacific/Chuuk"*

*[529] "Pacific/Easter"*

*[530] "Pacific/Efate"*

*[531] "Pacific/Enderbury"*

*[532] "Pacific/Fakaofo"*

*[533] "Pacific/Fiji"*

*[534] "Pacific/Funafuti"*

*[535] "Pacific/Galapagos"*

*[536] "Pacific/Gambier"*

*[537] "Pacific/Guadalcanal"*

*[538] "Pacific/Guam"*

*[539] "Pacific/Honolulu"*

*[540] "Pacific/Johnston"*

*[541] "Pacific/Kiritimati"*

*[542] "Pacific/Kosrae"*

*[543] "Pacific/Kwajalein"*

*[544] "Pacific/Majuro"*

*[545] "Pacific/Marquesas"*

*[546] "Pacific/Midway"*

*[547] "Pacific/Nauru"*

*[548] "Pacific/Niue"*

*[549] "Pacific/Norfolk"*

*[550] "Pacific/Noumea"*

*[551] "Pacific/Pago\_Pago"*

*[552] "Pacific/Palau"*

*[553] "Pacific/Pitcairn"*

*[554] "Pacific/Pohnpei"*

*[555] "Pacific/Ponape"*

*[556] "Pacific/Port\_Moresby"*

*[557] "Pacific/Rarotonga"*

*[558] "Pacific/Saipan"*

*[559] "Pacific/Samoa"*

*[560] "Pacific/Tahiti"*

*[561] "Pacific/Tarawa"*

*[562] "Pacific/Tongatapu"*

*[563] "Pacific/Truk"*

*[564] "Pacific/Wake"*

*[565] "Pacific/Wallis"*

*[566] "Pacific/Yap"*

*[567] "Poland"*

*[568] "Portugal"*

*[569] "PRC"*

*[570] "PST8PDT"*

*[571] "ROC"*

*[572] "ROK"*

*[573] "Singapore"*

*[574] "Turkey"*

*[575] "UCT"*

*[576] "Universal"*

*[577] "US/Alaska"*

*[578] "US/Aleutian"*

*[579] "US/Arizona"*

*[580] "US/Central"*

*[581] "US/East-Indiana"*

*[582] "US/Eastern"*

*[583] "US/Hawaii"*

*[584] "US/Indiana-Starke"*

*[585] "US/Michigan"*

*[586] "US/Mountain"*

*[587] "US/Pacific"*

*[588] "US/Pacific-New"*

*[589] "US/Samoa"*

*[590] "UTC"*

*[591] "W-SU"*

*[592] "WET"*

*[593] "Zulu"*

*attr(,"Version")*

*[1] "2019a"*

Lubridate also has function to extract hours, minutes and seconds form a time. We’ll show you some examples here:

*now()%>%hour()*

*[1] 13*

*> now()%>%minute()*

*[1] 37*

*> now()%>%second()*

*[1] 56.94769*

It also has functions that can parse strings into times. Let’s see some examples. Here’s an example:

*x<-"Nov/2/2012 12:34:56"*

*> mdy\_hms(x)*

*[1] "2012-11-02 12:34:56 UTC"*

This one takes a string that has both date and time and parses it our into a more appropriate format.

### Text Mining

Apart from labels used to represent categorical data, we have focused on numerical data, but in many applications data starts as text. Well known examples are spam filtering, cyber-crime prevention, counterterrorism and sentiment analysis.

In all these examples, the raw data is composed of free form texts. Our task is to extract insights from these data. In this section, we learn how to generate useful numerical summaries from text data to which we can apply some of the powerful data visualization and analysis techniques we have learned.

Case study: Trump Tweets

During the 2016 US presidential election, then-candidate Donald J. Trump used his Twitter account as a way to communicate with potential voters. On August 6, 2016 Todd Vaziri tweeted about Trump that "Every non-hyperbolic tweet is from iPhone (his staff). Every hyperbolic tweet is from Android (from him)." Data scientist David Robinson conducted an analysis to determine if data supported this assertion. Here we go through David's analysis to learn some of the basics of text mining. To learn more about text mining in R we recommend this book.

We will use the following libraries

*library(tidyverse)*

*library(ggplot2)*

*library(lubridate)*

*library(tidyr)*

*library(scales)*

*set.seed(1)*

In general, we can extract data directly from Twitter using the \emph{rtweet} package. However, in this case, a group has already compiled data for us and made it available at http://www.trumptwitterarchive.com.

*url <- 'http://www.trumptwitterarchive.com/data/realdonaldtrump/%s.json'*

*trump\_tweets <- map(2009:2017, ~sprintf(url, .x)) %>%*

*map\_df(jsonlite::fromJSON, simplifyDataFrame = TRUE) %>%*

*filter(!is\_retweet & !str\_detect(text, '^"')) %>%*

*mutate(created\_at = parse\_date\_time(created\_at, orders = "a b! d! H!:M!:S! z!\* Y!", tz="EST"))*

For convenience we include the result of the code above in the dslabs package:

*library(dslabs)*

*data("trump\_tweets")*

This is data frame with information about the tweet:

*head(trump\_tweets)*

The variables that are included are

*names(trump\_tweets)*

The help file ?trump\_tweets provides details on what each variable represents. The tweets are represented by the textvariable:

*trump\_tweets %>% select(text) %>% head*

and the source variable tells us the device that was used to compose and upload each tweet:

*trump\_tweets %>% count(source) %>% arrange(desc(n))*

We can use extract to remove the Twitter for part of the source and filter out retweets.

*trump\_tweets %>%*

*extract(source, "source", "Twitter for (.\*)") %>%*

*count(source)*

We are interested in what happened during the campaign, so for the analysis here we will focus on what was tweeted between the day Trump announced his campaign and election day. So we define the following table:

*campaign\_tweets <- trump\_tweets %>%*

*extract(source, "source", "Twitter for (.\*)") %>%*

*filter(source %in% c("Android", "iPhone") &*

*created\_at >= ymd("2015-06-17") &*

*created\_at < ymd("2016-11-08")) %>%*

*filter(!is\_retweet) %>%*

*arrange(created\_at)*

We can now use data visualization to explore the possibility that two different groups were tweeting from these devices. For each tweet, we will extract the hour, in the east coast (EST), it was tweeted then compute the proportion of tweets tweeted at each hour for each device.

*ds\_theme\_set()*

*campaign\_tweets %>%*

*mutate(hour = hour(with\_tz(created\_at, "EST"))) %>%*

*count(source, hour) %>%*

*group\_by(source) %>%*

*mutate(percent = n / sum(n)) %>%*

*ungroup %>%*

*ggplot(aes(hour, percent, color = source)) +*

*geom\_line() +*

*geom\_point() +*

*scale\_y\_continuous(labels = percent\_format()) +*

*labs(x = "Hour of day (EST)",*

*y = "% of tweets",*

*color = "")*

We notice a big peak for the Android in early hours of the morning, between 6 and 8 AM. There seems to be a clear different in these patterns. We will therefore assume that two different entities are using these two devices. Now we will study how their tweets differ. To do this we introduce the tidytext package.

Text as data

The tidytext package helps us convert free from text into a tidy table. Having the data in this format greatly facilitates data visualization and applying statistical techniques.

*Library(tidytext)*

The main function needed to achieve this is unnest\_tokens. A token refers to the units that we are considering to be a data point. The most common token will be words, but they can also be single characters, ngrams, sentences, lines or a pattern defined by a regex. The functions will take a vector of strings and extract the tokens so that each one gets a row in the new table. Here is a simple example:

*example <- data\_frame(line = c(1, 2, 3, 4),*

*text = c("Roses are red,", "Violets are blue,", "Sugar is sweet,", "And so are you."))*

*example*

*example %>% unnest\_tokens(word, text)*

Now let's look at a quick example with a tweet number 3008:

*i <- 3008*

*campaign\_tweets$text[i]*

*campaign\_tweets[i,] %>%*

*unnest\_tokens(word, text) %>%*

*select(word)*

Note that the function tries to convert tokens into words and strips characters important to twitter such as # and @. A token in twitter is not the same as in regular English. For this reason, instead of using the default token, words, we define a regex that captures twitter character. The pattern appears complex but all we are defining is a patter that starts with @, # or neither and is followed by any combination of letters or digits:

*pattern <- "([^A-Za-z\\d#@']|'(?![A-Za-z\\d#@]))"*

We can now use the unnest\_tokens function with the regex option and appropriately extract the hashtags and mentions:

*campaign\_tweets[i,] %>%*

*unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%*

*select(word)*

Another minor adjustment we want to make is remove the links to pictures:

*campaign\_tweets[i,] %>%*

*mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%*

*unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%*

*select(word)*

Now we are ready to extract the words for all our tweets.

tweet\_words <- campaign\_tweets %>%

mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern)

And we can now answer questions such as "what are the most commonly used words?"

*tweet\_words %>%*

*count(word) %>%*

*arrange(desc(n))*

It is not surprising that these are the top words. The top words are not informative. The tidytext package has database of these commonly used words, referred to as stop words, in text mining:

*stop\_words*

If we filter out rows representing stop words with filter(!word %in% stop\_words$word):

*tweet\_words <- campaign\_tweets %>%*

*mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%*

*unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%*

*filter(!word %in% stop\_words$word )*

We end up with a much more informative set of top 10 tweeted words:

*tweet\_words %>%*

*count(word) %>%*

*top\_n(10, n) %>%*

*mutate(word = reorder(word, n)) %>%*

*arrange(desc(n))*

Some exploration of the resulting words (not show here) reveals a couple of unwanted characteristics in our tokens. First, some of our tokens are just numbers (years for example). We want to remove these and we can find them using the regex ^\d+$. Second, some of our tokens come from a quote and they start with '. We want to remove the ' when it's at the start of a word, so we will use str\_replace. We add these two lines to the code above to generate our final table:

*tweet\_words <- campaign\_tweets %>%*

*mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%*

*unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%*

*filter(!word %in% stop\_words$word &*

*!str\_detect(word, "^\\d+$")) %>%*

*mutate(word = str\_replace(word, "^'", ""))*

Now that we have all our words in a table, along with information about what device was used to compose the tweet they came from, we can start exploring which words are more common when comparing Android to iPhone.

For each word we want to know if it is more likely to come from an Android tweet or an iPhone tweet. We previously introduced the odds ratio, a summary statistic useful for quantifying these differences. For each device and a given word, let's call it y, we compute the odds or the ratio between the proportion of words that are y and not y and compute the ratio of those odds. Here we will have many proportions that are 0 so we use the 0.5 correction.

*android\_iphone\_or <- tweet\_words %>%*

*count(word, source) %>%*

*spread(source, n, fill = 0) %>%*

*mutate(or = (Android + 0.5) / (sum(Android) - Android + 0.5) /*

*( (iPhone + 0.5) / (sum(iPhone) - iPhone + 0.5)))*

*android\_iphone\_or %>% arrange(desc(or))*

*android\_iphone\_or %>% arrange(or)*

Given that several of these words are overall low frequency words we can impose a filter based on the total frequency like this:

*android\_iphone\_or %>% filter(Android+iPhone > 100) %>%*

*arrange(desc(or))*

*android\_iphone\_or %>% filter(Android+iPhone > 100) %>%*

*arrange(or)*

We already see somewhat of a pattern in the types of words that are being tweeted more in one device versus the other. However, we are not interested in specific words but rather in the tone. Vaziri's assertion is that the Android tweets are more hyperbolic. So how can we check this with data? Hyperbolic is a hard sentiment to extract from words as it relies on interpreting phrases. However, words can be associated to more basic sentiment such as as anger, fear, joy and surprise. In the next section we demonstrate basic sentiment analysis.

Sentiment Analysis

In sentiment analysis we assign a word to one or more "sentiment". Although this approach will miss context dependent sentiments, such as sarcasm, when performed on large numbers of words, summaries can provide insights.

The first step in sentiment analysis is to assign a sentiment to each word. The tidytext package includes several maps or lexicons in the object sentiments:

sentiments

There are several lexicons in the tidytext package that give different sentiments. For example, the bing lexicon divides words into positive and negative. We can see this using the tidytext function get\_sentiments:

*get\_sentiments("bing")*

The AFINN lexicon assigns a score between -5 and 5, with -5 the most negative and 5 the most positive.

*get\_sentiments("afinn")*

*The loughran and nrc lexicons provide several different sentiments:*

*get\_sentiments("loughran") %>% count(sentiment)*

*get\_sentiments("nrc") %>% count(sentiment)*

To start learning about how these lexicons were developed, read this help file:

*?sentiments.*

For the analysis here we are interested in exploring the different sentiments of each tweet, so we will use the nrc lexicon.

*nrc <- get\_sentiments("nrc")*

We can combine the words and sentiments using inner\_join, which will only keep words associated with a sentiment. Here are 10 random words extracted from the tweets:

*tweet\_words %>% inner\_join(nrc, by = "word") %>%*

*select(source, word, sentiment) %>% sample\_n(10)*

Now we are ready to perform a quantitative analysis comparing Android and iPhone by comparing the sentiments of the tweets posted from each device. Here we could perform a tweet by tweet analysis, assigning a sentiment to each tweet. However, this somewhat complex since each tweet will have several sentiments attached to it, one for each word appearing in the lexicon. For illustrative purposes, we will perform a much simpler analysis: we will count and compare the frequencies of each sentiment appears for each device.

*sentiment\_counts <- tweet\_words %>%*

*left\_join(nrc, by = "word") %>%*

*count(source, sentiment) %>%*

*spread(source, n) %>%*

*mutate(sentiment = replace\_na(sentiment, replace = "none"))*

*sentiment\_counts*

Because more words were used on the Android than on the phone:

*tweet\_words %>% group\_by(source) %>% summarize(n = n())*

for each sentiment we can compute the odds of being in the device: proportion of words with sentiment versus proportion of words without and then compute the odds ratio comparing the two devices:

*sentiment\_counts %>%*

*mutate(Android = Android / (sum(Android) - Android) ,*

*iPhone = iPhone / (sum(iPhone) - iPhone),*

*or = Android/iPhone) %>%*

*arrange(desc(or))*

So we do see some difference and the order is interesting: the largest three sentiments are disgust, anger, and negative! But are they statistically significant? How does this compare if we are just assigning sentiments at random?

To answer that question we can compute, for each sentiment, an odds ratio and confidence interval. We will add the two values we need to form a two-by-two table and the odds ratio:

*library(broom)*

*log\_or <- sentiment\_counts %>%*

*mutate( log\_or = log( (Android / (sum(Android) - Android)) / (iPhone / (sum(iPhone) - iPhone))),*

*se = sqrt( 1/Android + 1/(sum(Android) - Android) + 1/iPhone + 1/(sum(iPhone) - iPhone)),*

*conf.low = log\_or - qnorm(0.975)\*se,*

*conf.high = log\_or + qnorm(0.975)\*se) %>%*

*arrange(desc(log\_or))*

*log\_or*

A graphical visualization shows some sentiments that are clearly overrepresented:

*log\_or %>%*

*mutate(sentiment = reorder(sentiment, log\_or),) %>%*

*ggplot(aes(x = sentiment, ymin = conf.low, ymax = conf.high)) +*

*geom\_errorbar() +*

*geom\_point(aes(sentiment, log\_or)) +*

*ylab("Log odds ratio for association between Android and sentiment") +*

*coord\_flip()*

We see that the disgust, anger, negative sadness and fear sentiments are associated with the Android in a way that is hard to explain by chance alone. Words not associated to a sentiment were strongly associated with the iPhone source, which is in agreement with the original claim about hyperbolic tweets.

If we are interested in exploring which specific words are driving these differences, we can back to our android\_iphone\_or object:

*android\_iphone\_or %>% inner\_join(nrc) %>%*

*filter(sentiment == "disgust" & Android + iPhone > 10) %>%*

*arrange(desc(or))*

We can make a graph:

*android\_iphone\_or %>% inner\_join(nrc, by = "word") %>%*

*mutate(sentiment = factor(sentiment, levels = log\_or$sentiment)) %>%*

*mutate(log\_or = log(or)) %>%*

*filter(Android + iPhone > 10 & abs(log\_or)>1) %>%*

*mutate(word = reorder(word, log\_or)) %>%*

*ggplot(aes(word, log\_or, fill = log\_or < 0)) +*

*facet\_wrap(~sentiment, scales = "free\_x", nrow = 2) +*

*geom\_bar(stat="identity", show.legend = FALSE) +*

*theme(axis.text.x = element\_text(angle = 90, hjust = 1))*