**Data Science: Wrangling**

[**https://github.com/1965Eric/HarvardX-PH125.6x-Data-Science-Wrangling/blob/master/Data\_Science\_Wrangling.Rmd**](https://github.com/1965Eric/HarvardX-PH125.6x-Data-Science-Wrangling/blob/master/Data_Science_Wrangling.Rmd)

[**https://rpubs.com/faisalcep/dsWrangling**](https://rpubs.com/faisalcep/dsWrangling)

[**https://www.gutenberg.org/MIRRORS.ALL**](https://www.gutenberg.org/MIRRORS.ALL)

[**https://1965eric.github.io/Wrangling/section-3-overview.html**](https://1965eric.github.io/Wrangling/section-3-overview.html)

**Key points**

* Many datasets are stored in spreadsheets. A spreadsheet is essentially a file version of a data frame with rows and columns.
* Spreadsheets have rows separated by returns and columns separated by a delimiter. The most common delimiters are comma, semicolon, white space and tab.
* Many spreadsheets are raw text files and can be read with any basic text editor. However, some formats are proprietary and cannot be read with a text editor, such as Microsoft Excel files (.xls).
* Most import functions assume that the first row of a spreadsheet file is a header with column names. To know if the file has a header, it helps to look at the file with a text editor before trying to import it.

**Key points**

* The working directory is where R looks for files and saves files by default.
* See your working directory with getwd(). Change your working directory with setwd().
* We suggest you create a directory for each project and keep your raw data inside that directory.
* Use the file.path() function to generate a full path from a relative path and a file name. Use file.path() instead of paste() because file.path() is aware of your operating system and will use the correct slashes to navigate your machine.
* The file.copy() function copies a file to a new path.

**Code**

# see working directory

getwd()

# change your working directory

setwd()

# set path to the location for raw data files in the dslabs package and list files

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

list.files(path)

# generate a full path to a file

filename <- "murders.csv"

fullpath <- file.path(path, filename)

fullpath

# copy file from dslabs package to your working directory

file.copy(fullpath, getwd())

# check if the file exists

file.exists(filename)

### Key points

* **readr**is the **tidyverse** library that includes functions for reading data stored in text file spreadsheets into R. Functions in the package include read\_csv(), read\_tsv(), read\_delim() and more. These differ by the delimiter they use to split columns.
* The **readxl** package provides functions to read Microsoft Excel formatted files.
* The excel\_sheets() function gives the names of the sheets in the Excel file. These names are passed to the sheet argument for the **readxl** functions read\_excel(), read\_xls() and read\_xlsx().
* The read\_lines() function shows the first few lines of a file in R.

### Code

library(dslabs)

library(tidyverse) # includes readr

library(readxl)

# inspect the first 3 lines

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

# read file in CSV format

dat <- read\_csv(filename)

#read using full path

dat <- read\_csv(fullpath)

head(dat)

#Ex：

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

files <- list.files(path)

files

filename <- "murders.csv"

filename1 <- "life-expectancy-and-fertility-two-countries-example.csv"

filename2 <- "fertility-two-countries-example.csv"

dat=read.csv(file.path(path, filename))

dat1=read.csv(file.path(path, filename1))

dat2=read.csv(file.path(path, filename2))

### Key points

* R-base import functions (read.csv(), read.table(), read.delim()) generate data frames rather than tibbles.
* Note that as of R 4.0, it is no longer necessary to use the argument stringsAsFactors=FALSE to prevent characters from being converted into factors.

### Code

# filename is defined in the previous video

# read.csv to import the data

dat2 <- read.csv(filename)

class(dat2$abb)

class(dat2$region)

print.dat.frame(dat2)

### Key points

* The read\_csv() function and other import functions can read a URL directly.
* If you want to have a local copy of the file, you can use download.file().
* tempdir() creates a directory with a name that is very unlikely not to be unique.
* tempfile() creates a character string that is likely to be a unique filename.

### Code

url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"

dat <- read\_csv(url)

download.file(url, "murders.csv")

tempfile()

tmp\_filename <- tempfile()

download.file(url, tmp\_filename)

dat <- read\_csv(tmp\_filename)

file.remove(tmp\_filename)

Leer desde Excel

times\_2016 <- read\_excel("times.xlsx", sheet = "2016")

### Key points

* In tidy data, each row represents an observation and each column represents a different variable.
* In wide data, each row includes several observations and one of the variables is stored in the header.

### Code

library(tidyverse)

library(dslabs)

data(gapminder)

# create and inspect a tidy data frame

tidy\_data <- gapminder %>%

filter(country %in% c("South Korea", "Germany")) %>%

select(country, year, fertility)

head(tidy\_data)

# plotting tidy data is simple

tidy\_data %>%

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

geom\_point()

# import and inspect example of original Gapminder data in wide format

path <- system.file("extdata", package="dslabs")  
filename <- file.path(path, "fertility-two-countries-example.csv")  
wide\_data <- read\_csv(filename)  
select(wide\_data, country, `1960`:`1967`)

### Key points

* After importing data, a common next step is to reshape the data into a form useful for the rest of the analysis by tidying it. The **tidyr** package includes several useful functions for tidying data.
* The pivot\_longer() function converts wide data into tidy data.
* The first argument of pivot\_longer() is the data frame to be reshaped. The second argument specifies the columns containing the values to be moved into a single column.
* The new column of values is called value by default and the column containing the original names of those columns is called name by default.
* The values\_to and names\_to arguments can be used to change the default names of these columns.

### Code

# example dataset: fertility data in wide format (from previous video)

library(tidyverse)

library(dslabs)

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

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

wide\_data <- read\_csv(filename)

# snippet of wide data

wide\_data %>% select(country, '1960':'1965')

# move the values in the columns 1960 through 2015 into a single column

wide\_data %>% pivot\_longer(`1960`:`2015`)

# another way to do this - only country isn't being pivoted

wide\_data %>% pivot\_longer(-country)

# change the default column names

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility")

head(new\_tidy\_data)

# compare the class from our original tidy data (year is an integer) and in the new version (year is a character)

class(tidy\_data$year)

class(new\_tidy\_data$year)

# use the names\_transform argument to change the class of the year values to numeric

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility",

names\_transform = list(year=as.numeric))

# plot the data as before

new\_tidy\_data %>% ggplot(aes(year, fertility, color = country)) +

geom\_point()

### Key points

* The pivot\_wider() function converts tidy data into wide data, which can be a useful intermediate step in data tidying.
* The data frame to be reshaped is the first argument in pivot\_wider().
* The argument names\_from tells pivot\_wider() which variable will be used for the column names and the argument values\_from tells pivot\_wider() which variable to use to fill in the values.
* The [tidyr cheat sheet External link](https://github.com/rstudio/cheatsheets/blob/main/tidyr.pdf) is a useful reference for these and other functions.

### Code

# still working with the same data as in the previous video

# convert the tidy data to wide data

new\_wide\_data <- new\_tidy\_data %>%

pivot\_wider(names\_from = year, values\_from = fertility)

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

### Key points

* The separate() function splits one column into two or more columns at a specified character that separates the variables.
* The separate() function takes three arguments (apart from the data): the name of the column to be separated, the names to be used for the new columns, and the character that separates the variables.
* When there is an extra separation, you can use extra = "merge" to merge the last two variables.

### Code

# import data

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

fname <- "life-expectancy-and-fertility-two-countries-example.csv"

filename <- file.path(path, fname)

raw\_dat <- read\_csv(filename)

select(raw\_dat, 1:4)

# pivot all columns except country

dat <- raw\_dat %>% pivot\_longer(-country)

head(dat)

dat$name[1:5]

# separate on underscores

dat %>% separate(name, c("year", "name"), sep = "\_")

# separate on underscores (the default), convert years to numeric

dat %>% separate(name, c("year", "name"), convert = TRUE)

# split on all underscores, pad empty cells with NA

dat %>% separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE)

# split on first underscore but keep life\_expectancy merged

dat %>% separate(name, c("year", "name"), sep = "\_",

extra = "merge", convert = TRUE)

# separate then create a new column for each variable using pivot\_wider

dat %>% separate(name, c("year", "name"), sep = "\_",

extra = "merge", convert = TRUE) %>%

pivot\_wider()

### Key points

* The unite() function joins two columns into one.

### Code

# using the data from the previous video

# if we had used this non-optimal approach to separate

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE)

# we could unite the second and third columns using unite()

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE) %>%

unite(variable\_name, name\_1, name\_2, sep="\_")

# spread the columns

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE) %>%

unite(name, name\_1, name\_2, sep="\_") %>%

spread(name, value) %>%

rename(fertlity = fertility\_NA)

### Key points

* The join functions in the **dplyr** package combine two tables such that matching rows are together.
* left\_join() only keeps rows that have information in the first table.
* right\_join() only keeps rows that have information in the second table.
* inner\_join() only keeps rows that have information in both tables.
* full\_join() keeps all rows from both tables.
* semi\_join() keeps the part of first table for which we have information in the second.
* anti\_join() keeps the elements of the first table for which there is no information in the second.

### Code

# import US murders data

library(tidyverse)  
library(ggrepel)  
library(dslabs)  
ds\_theme\_set()  
data(murders)  
head(murders)

# import US election results data

data(polls\_us\_election\_2016)  
head(results\_us\_election\_2016)  
identical(results\_us\_election\_2016$state, murders$state)

# join the murders table and US election results table

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

# plot electoral votes versus population

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)

# make two smaller tables to demonstrate joins

tab1 <- slice(murders, 1:6) %>% select(state, population)  
tab1  
tab2 <- slice(results\_us\_election\_2016, c(1:3, 5, 7:8)) %>% select(state, electoral\_votes)  
tab2

# experiment with different joins

left\_join(tab1, tab2)  
tab1 %>% left\_join(tab2)  
tab1 %>% right\_join(tab2)  
inner\_join(tab1, tab2)  
semi\_join(tab1, tab2)  
anti\_join(tab1, tab2)

### Key points

* Unlike the join functions, the binding functions do not try to match by a variable, but rather just combine datasets.
* bind\_cols() binds two objects by making them columns in a tibble. The R-base function cbind() binds columns but makes a data frame or matrix instead.
* The bind\_rows() function is similar but binds rows instead of columns. The R-base function rbind() binds rows but makes a data frame or matrix instead.

### Code

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

tab1 <- tab[, 1:3]  
tab2 <- tab[, 4:6]  
tab3 <- tab[, 7:9]  
new\_tab <- bind\_cols(tab1, tab2, tab3)  
head(new\_tab)

tab1 <- tab[1:2,]  
tab2 <- tab[3:4,]  
bind\_rows(tab1, tab2)

### Key points

* By default, the set operators in R-base work on vectors. If **tidyverse/dplyr** are loaded, they also work on data frames.
* You can take intersections of vectors using intersect(). This returns the elements common to both sets.
* You can take the union of vectors using union(). This returns the elements that are in either set.
* The set difference between a first and second argument can be obtained with setdiff(). Note that this function is not symmetric.
* The function set\_equal() tells us if two sets are the same, regardless of the order of elements.

### Code

# intersect vectors or data frames

intersect(1:10, 6:15)  
intersect(c("a","b","c"), c("b","c","d"))  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
intersect(tab1, tab2)

# perform a union of vectors or data frames

union(1:10, 6:15)  
union(c("a","b","c"), c("b","c","d"))  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
union(tab1, tab2)

# set difference of vectors or data frames

setdiff(1:10, 6:15)  
setdiff(6:15, 1:10)  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
setdiff(tab1, tab2)

# setequal determines whether sets have the same elements, regardless of order

setequal(1:5, 1:6)  
setequal(1:5, 5:1)  
setequal(tab1, tab2)

### Key points

* Web scraping is extracting data from a website.
* The **rvest** web harvesting package includes functions to extract nodes of an HTML document: html\_nodes() extracts all nodes of different types, and html\_node() extracts the first node.
* html\_table() converts an HTML table to a data frame.

### Code

# import a webpage into R

library(rvest)  
url <- "https://en.wikipedia.org/wiki/Murder\_in\_the\_United\_States\_by\_state"  
h <- read\_html(url)  
class(h)  
h

tab <- h %>% html\_nodes("table")  
tab <- tab[[2]]

tab <- tab %>% html\_table  
class(tab)

tab <- tab %>% setNames(c("state", "population", "total", "murders", "gun\_murders", "gun\_ownership", "total\_rate", "murder\_rate", "gun\_murder\_rate"))  
head(tab)

# 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 External link](http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609). Looking at the code for this page, it seems that the task is impossibly complex. However, selector gadgets actually make this possible. [SelectorGadget External link](http://selectorgadget.com/) 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.

### Key points

* The most common tasks in string processing include:
  + - extracting numbers from strings
    - removing unwanted characters from text
    - finding and replacing characters
    - extracting specific parts of strings
    - converting free form text to more uniform formats
    - splitting strings into multiple values
* The **stringr** package in the **tidyverse** contains string processing functions that follow a similar naming format (str\_functionname) and are compatible with the pipe.

### Code

# read in raw murders data from Wikipedia

url <- "https://en.wikipedia.org/w/index.php?title=Gun\_violence\_in\_the\_United\_States\_by\_state&direction=prev&oldid=810166167"  
murders\_raw <- read\_html(url) %>%   
 html\_nodes("table") %>%   
 html\_table() %>%  
 .[[1]] %>%  
 setNames(c("state", "population", "total", "murder\_rate"))

# inspect data and column classes

head(murders\_raw)  
class(murders\_raw$population)  
class(murders\_raw$total)

### Key points

* Define a string by surrounding text with either single quotes or double quotes.
* To include a single quote inside a string, use double quotes on the outside. To include a double quote inside a string, use single quotes on the outside.
* The cat() function displays a string as it is represented inside R.
* To include a double quote inside of a string surrounded by double quotes, use the backslash (\) to escape the double quote. Escape a single quote to include it inside of a string defined by single quotes.
* We will see additional uses of the escape later.

### Code

s <- "Hello!" # double quotes define a string

s <- 'Hello!' # single quotes define a string

s <- `Hello` # backquotes do not

s <- "10"" # error - unclosed quotes

s <- '10"' # correct

# cat shows what the string actually looks like inside R

cat(s)

s <- "5'"  
cat(s)

# to include both single and double quotes in string, escape with \

s <- '5'10"' # error

s <- "5'10"" # error

s <- '5\'10"' # correct

cat(s)

s <- "5'10\"" # correct

cat(s)

### Key points

* The main types of string processing tasks are detecting, locating, extracting and replacing elements of strings.
* The **stringr** package from the **tidyverse** includes a variety of string processing functions that begin with str\_ and take the string as the first argument, which makes them compatible with the pipe.

### Code

# murders\_raw defined in web scraping video

# direct conversion to numeric fails because of commas

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

library(tidyverse) # includes stringr

### Key points

* Use the str\_detect() function to determine whether a string contains a certain pattern.
* Use the str\_replace\_all() function to replace all instances of one pattern with another pattern. To remove a pattern, replace with the empty string ("").
* The parse\_number() function removes punctuation from strings and converts them to numeric.
* mutate\_at() performs the same transformation on the specified column numbers.

### Code

# murders\_raw was defined in the web scraping section

# detect whether there are commas

commas <- function(x) any(str\_detect(x, ","))  
murders\_raw %>% summarize\_all(funs(commas))

# replace commas with the empty string and convert to numeric

test\_1 <- str\_replace\_all(murders\_raw$population, ",", "")  
test\_1 <- as.numeric(test\_1)

# parse\_number also removes commas and converts to numeric

test\_2 <- parse\_number(murders\_raw$population)  
identical(test\_1, test\_2)

murders\_new <- murders\_raw %>% mutate\_at(2:3, parse\_number)  
murders\_new %>% head

### Key points

* In the raw heights data, many students did not report their height as the number of inches as requested. There are many entries with real height information but in the wrong format, which we can extract with string processing.
* When there are both text and numeric entries in a column, the column will be a character vector. Converting this column to numeric will result in NAs for some entries.
* To correct problematic entries, look for patterns that are shared across large numbers of entries, then define rules that identify those patterns and use these rules to write string processing tasks.
* Use suppressWarnings() to hide warning messages for a function.

### Code

# load raw heights data and inspect

library(dslabs)  
data(reported\_heights)  
class(reported\_heights$height)

# convert to numeric, inspect, count NAs

x <- as.numeric(reported\_heights$height)  
head(x)  
sum(is.na(x))

# keep only entries that result in NAs

reported\_heights %>% mutate(new\_height = as.numeric(height)) %>%  
 filter(is.na(new\_height)) %>%   
 head(n=10)

# calculate cutoffs that cover 99.999% of human population

alpha <- 1/10^6  
qnorm(1-alpha/2, 69.1, 2.9)  
qnorm(alpha/2, 63.7, 2.7)

# keep only entries that either result in NAs or are outside the plausible range of heights

not\_inches <- function(x, smallest = 50, tallest = 84){  
 inches <- suppressWarnings(as.numeric(x))  
 ind <- is.na(inches) | inches < smallest | inches > tallest  
 ind  
}

# number of problematic entries

problems <- reported\_heights %>%   
 filter(not\_inches(height)) %>%  
 .$height  
length(problems)

# 10 examples of x'y or x'y" or x'y\"

pattern <- "^\\d\\s\*'\\s\*\\d{1,2}\\.\*\\d\*'\*\"\*$"  
str\_subset(problems, pattern) %>% head(n=10) %>% cat

# 10 examples of x.y or x,y

pattern <- "^[4-6]\\s\*[\\.|,]\\s\*([0-9]|10|11)$"  
str\_subset(problems, pattern) %>% head(n=10) %>% cat

# 10 examples of entries in cm rather than inches

ind <- which(between(suppressWarnings(as.numeric(problems))/2.54, 54, 81) )  
ind <- ind[!is.na(ind)]  
problems[ind] %>% head(n=10) %>% cat

### Key points

* A regular expression (regex) is a way to describe a specific pattern of characters of text. A set of rules has been designed to do this specifically and efficiently.
* **stringr** functions can take a regex as a pattern.
* str\_detect() indicates whether a pattern is present in a string.
* The main difference between a regex and a regular string is that a regex can include special characters.
* The | symbol inside a regex means "or".
* Use '\\d' to represent digits. The backlash is used to distinguish it from the character 'd'. In R, you must use two backslashes for digits in regular expressions; in some other languages, you will only use one backslash for regex special characters.
* str\_view() highlights the first occurrence of a pattern, and the str\_view\_all() function highlights all occurrences of the pattern.

### Code

# load stringr through tidyverse

library(tidyverse)

# detect whether a comma is present

pattern <- ","  
str\_detect(murders\_raw$total, pattern)

# show the subset of strings including "cm"

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

# use the "or" symbol inside a regex (|)

yes <- c("180 cm", "70 inches")  
no <- c("180", "70''")  
s <- c(yes, no)  
str\_detect(s, "cm") | str\_detect(s, "inches")  
str\_detect(s, "cm|inches")

# highlight the first occurrence of a pattern

str\_view(s, pattern)

# highlight all instances of a pattern

str\_view\_all(s, pattern)

### Key points

* Define strings to test your regular expressions, including some elements that match and some that do not. This allows you to check for the two types of errors: failing to match and matching incorrectly.
* Square brackets define character classes: groups of characters that count as matching the pattern. You can use ranges to define character classes, such as [0-9] for digits and [a-zA-Z] for all letters.
* Anchors define patterns that must start or end at specific places. ^ and $ represent the beginning and end of the string respectively.
* Curly braces are quantifiers that state how many times a certain character can be repeated in the pattern. \\d{1,2} matches exactly 1 or 2 consecutive digits.

### Code

# s was defined in the previous video

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

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

s <- c(yes, no)

pattern <- "\\d"

# [56] means 5 or 6

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

# [4-7] means 4, 5, 6 or 7

yes <- as.character(4:7)  
no <- as.character(1:3)  
s <- c(yes, no)  
str\_detect(s, "[4-7]")

# ^ means start of string, $ means end of string

pattern <- "^\\d$"  
yes <- c("1", "5", "9")  
no <- c("12", "123", " 1", "a4", "b")  
s <- c(yes, no)  
str\_view(s, pattern)

# curly braces define quantifiers: 1 or 2 digits

pattern <- "^\\d{1,2}$"  
yes <- c("1", "5", "9", "12")  
no <- c("123", "a4", "b")  
str\_view(c(yes, no), pattern)

# combining character class, anchors and quantifier

pattern <- "^[4-7]'\\d{1,2}\"$"  
yes <- c("5'7\"", "6'2\"", "5'12\"")  
no <- c("6,2\"", "6.2\"","I am 5'11\"", "3'2\"", "64")  
str\_detect(yes, pattern)  
str\_detect(no, pattern)

### Key points

* str\_replace() replaces the first instance of the detected pattern with a specified string.
* Spaces are characters and R does not ignore them. Spaces are specified by the special character \\s.
* Additional quantifiers include \*, + and ?. \* means 0 or more instances of the previous character. ? means 0 or 1 instances. + means 1 or more instances.
* Before removing characters from strings with functions like str\_replace() and str\_replace\_all(), consider whether that replacement would have unintended effects.

### Code

The problems object is defined in the [reported heights case study introduction video](https://courses.edx.org/courses/course-v1:HarvardX+PH125.6x+2T2019/courseware/c59e9550f970406e81b8a908ce42dcc0/58fca697d7e2436187ff7059588d94e8/1?activate_block_id=block-v1%3AHarvardX%2BPH125.6x%2B2T2019%2Btype%40vertical%2Bblock%402cc7ac16174f4736aac210300b8179ba).

# number of entries matching our desired pattern

pattern <- "^[4-7]'\\d{1,2}\"$"  
sum(str\_detect(problems, pattern))

# inspect examples of entries with problems

problems[c(2, 10, 11, 12, 15)] %>% str\_view(pattern)  
str\_subset(problems, "inches")  
str\_subset(problems, "''")

# replace or remove feet/inches words before matching

pattern <- "^[4-7]'\\d{1,2}$"  
problems %>%   
 str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

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

str\_detect(pattern) %>%   
 sum()

# R does not ignore whitespace

identical("Hi", "Hi ")

# \\s represents whitespace

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

str\_subset(problems, pattern\_2)

# \* means 0 or more instances of a character

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

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

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

# test how \*, ? and + differ

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

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

nore\_or\_once = str\_detect(yes, "A1?B"),

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

# update pattern by adding optional spaces before and after feet symbol

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

problems %>%

str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

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

str\_detect(pattern) %>%

sum()

### Key Points

* Groups are defined using parentheses.
* Once we define groups, we can use the function str\_match() to extract the values these groups define. str\_extract() extracts only strings that match a pattern, not the values defined by groups.
* You can refer to the ith group with \\i. For example, refer to the value in the second group with \\2.

### Code

# define regex with and without groups

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

# create examples

yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)

# demonstrate the effect of groups

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

# demonstrate difference between str\_match and str\_extract

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

# improve the pattern to recognize more events

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")

# final pattern

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

# combine stringr commands with the pipe

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

# define regex with and without groups

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

# create examples

yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)

# demonstrate the effect of groups

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

# demonstrate difference between str\_match and str\_extract

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

# improve the pattern to recognize more events

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")

# final pattern

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

# combine stringr commands with the pipe

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

### Key Point

* The extract() function behaves similarly to the separate() function but allows extraction of groups from regular expressions.

### Code

# first example - normally formatted heights

s <- c("5'10", "6'1")  
tab <- data.frame(x = s)

# the separate and extract functions behave similarly

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

# second example - some heights with unusual formats

s <- c("5'10", "6'1\"","5'8inches")  
tab <- data.frame(x = s)

# separate fails because it leaves in extra characters, but extract keeps only the digits because of regex groups

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

# 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 entires 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]

### Key Points

* The function str\_split() splits a string into a character vector on a delimiter (such as a comma, space or underscore). By default, str\_split() generates a list with one element for each original string. Use the function argument simplify=TRUE to have str\_split() return a matrix instead.
* The map() function from the **purrr** package applies the same function to each element of a list. To extract the ith entry of each element x, use map(x, i).
* map() always returns a list. Use map\_chr() to return a character vector and map\_int() to return an integer.

### Code

# read raw murders data line by line

filename <- system.file("extdata/murders.csv", package = "dslabs")  
lines <- readLines(filename)  
lines %>% head()

# split at commas with str\_split function, remove row of column names

x <- str\_split(lines, ",")   
x %>% head()  
col\_names <- x[[1]]  
x <- x[-1]

# extract first element of each list entry

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

# extract columns 1-5 as characters, then convert to proper format - NOTE: DIFFERENT FROM VIDEO

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

# more efficient code for the same thing

dat <- x %>%  
 transpose() %>%  
 map( ~ parse\_guess(unlist(.))) %>%  
 setNames(col\_names) %>%   
 as.data.frame()

# the simplify argument makes str\_split return a matrix instead of a list

x <- str\_split(lines, ",", simplify = TRUE)   
col\_names <- x[1,]  
x <- x[-1,]  
x %>% as\_data\_frame() %>%  
 setNames(col\_names) %>%  
 mutate\_all(parse\_guess)

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

library(dslabs)

data("research\_funding\_rates")

research\_funding\_rates

The data come from a [paper External link](http://www.pnas.org/content/112/40/12349.abstract) 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 <- "https://www.pnas.org/action/downloadSupplement?doi=10.1073%2Fpnas.1510159112&file=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 actually 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 can 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 fourth 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

### Key points

* Change long factor names with the recode() function from the **tidyverse**.
* Other similar functions include recode\_factor() and fct\_recoder() in the **forcats** package in the **tidyverse**. The same result could be obtained using the case\_when() function, but recode() is more efficient to write.

### Code

# life expectancy time series for Caribbean countries

library(dslabs)  
data("gapminder")  
gapminder %>%   
 filter(region=="Caribbean") %>%  
 ggplot(aes(year, life\_expectancy, color = country)) +  
 geom\_line()

# display long country names

gapminder %>%   
 filter(region=="Caribbean") %>%  
 filter(str\_length(country) >= 12) %>%  
 distinct(country)

# recode long country names and remake plot

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' = "Trinidad")) %>%  
 ggplot(aes(year, life\_expectancy, color = country)) +  
 geom\_line()

### Key points

* Dates are a separate data type in R.The **tidyverse** includes functionality for dealing with dates through the **lubridate** package.
* Extract the year, month and day from a date object with the year(), month() and day() functions.
* Parsers convert strings into dates with the standard YYYY-MM-DD format (ISO 8601 format). Use the parser with the name corresponding to the string format of year, month and day (ymd(), ydm(), myd(), mdy(), dmy(), dym()).
* Get the current time with the Sys.time() function. Use the now() function instead to specify a time zone.
* You can extract values from time objects with the hour(), minute() and second() functions.
* Parsers convert strings into times (for example, hms()). Parsers can also create combined date-time objects (for example, mdy\_hms()).

### Code

# inspect the startdate column of 2016 polls data, a Date type

library(tidyverse)  
library(dslabs)  
data("polls\_us\_election\_2016")  
polls\_us\_election\_2016$startdate %>% head  
class(polls\_us\_election\_2016$startdate)  
as.numeric(polls\_us\_election\_2016$startdate) %>% head

# ggplot is aware of dates

polls\_us\_election\_2016 %>% filter(pollster == "Ipsos" & state =="U.S.") %>%  
 ggplot(aes(startdate, rawpoll\_trump)) +  
 geom\_line()

# lubridate: the tidyverse date package

library(lubridate)

# select some random dates from polls

set.seed(2)  
dates <- sample(polls\_us\_election\_2016$startdate, 10) %>% sort  
dates

# extract month, day, year from date strings

data.frame(date = dates,   
 month = month(dates),  
 day = day(dates),  
 year = year(dates))

month(dates, label = TRUE) # extract month label

# ymd works on mixed date styles

x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4",  
 "2009-1, 5", "Created on 2009 1 6", "200901 !!! 07")  
ymd(x)

# different parsers extract year, month and day in different orders

x <- "09/01/02"  
ymd(x)  
mdy(x)  
ydm(x)  
myd(x)  
dmy(x)  
dym(x)

now() # current time in your time zone

now("GMT") # current time in GMT

now() %>% hour() # current hour

now() %>% minute() # current minute

now() %>% second() # current second

# parse time

x <- c("12:34:56")  
hms(x)

#parse datetime

x <- "Nov/2/2012 12:34:56"  
mdy\_hms(x)

# Text Mining

### Key points

* The **tidytext** package helps us convert free form text into a tidy table.
* Use unnest\_tokens() to extract individual words and other meaningful chunks of text.
* Sentiment analysis assigns emotions or a positive/negative score to tokens. You can extract sentiments using get\_sentiments(). Common lexicons for sentiment analysis are "bing", "afinn", "nrc" and "loughran".

With the exception of 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, counter-terrorism 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 External link](https://twitter.com/tvaziri/status/762005541388378112/photo/1) 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 External link](http://varianceexplained.org/r/trump-tweets/) 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 External link](https://www.tidytextmining.com/).

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 **rtweet** package. However, in this case, a group has already compiled data for us and made it available at [https://www.thetrumparchive.com/ External link](https://www.thetrumparchive.com/).

url <- 'https://drive.google.com/file/d/16wm-2NTKohhcA26w-kaWfhLIGwl\_oX95/view'

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 text variable:

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 tokens 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") %>%

select(word, sentiment)

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))

---

title: "Data Science Wrangling"

output:

bookdown::epub\_book:

number\_sections: true

word\_document: default

html\_document: default

pdf\_document:

latex\_engine: xelatex

urlcolor: blue

always\_allow\_html: yes

---

```{r setup, include=FALSE}

knitr::opts\_chunk$set(echo = TRUE)

```

The textbook for the Data Science course series is [freely available online](https://rafalab.github.io/dsbook/){target="\_blank"}.

# Learning Objectives

\* How to import data into R from different file formats

\* How to scrape data from the web

\* How to tidy data using the tidyverse to better facilitate analysis

\* How to process strings with regular expressions (regex)

\* How to wrangle data using dplyr

\* How to work with dates and times as file formats

\* How to mine text

## Course Overview

### Section 1: Data Import

You will learn how to import data from different sources.

### Section 2: Tidy Data

You will learn the first pieces of converting data into a tidy format.

### Section 3: String Processing

You will learn how to process strings using regular expressions (regex).

### Section 4: Dates, Times, and Text Mining

You will learn how to work with dates and times as file formats and how to mine text.

## Introduction to Wrangling

The textbook for this section is available [here](https://rafalab.github.io/dsbook/introduction-to-data-wrangling.html){target="\_blank"}.

\*\*Key points\*\*

\* The first step in data analysis is importing, tidying and cleaning the data. This is the process of data wrangling.

\* In this course, we cover several common steps of the data wrangling process: tidying data, string processing, html parsing, working with dates and times, and text mining.

# Section 1 Overview

In the \*\*Data Import\*\* section, you will learn how import data into R.

After completing this section, you will be able to:

\* \*\*Import\*\* data from spreadsheets.

- Identify and set your \*\*working directory\*\* and specify the \*\*path\*\* to a file.

- Use the \*\*readr\*\* and \*\*readxl\*\* packages to import spreadsheets.

- Use \*\*R-base functions\*\* to import spreadsheets.

- \*\*Download\*\* files from the internet using R.

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html){target="\_blank"}.

## Importing Spreadsheets

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html){target="\_blank"}.

\*\*Key points\*\*

\* Many datasets are stored in spreadsheets. A spreadsheet is essentially a file version of a data frame with rows and columns.

\* Spreadsheets have rows separated by returns and columns separated by a delimiter. The most common delimiters are comma, semicolon, white space and tab.

\* Many spreadsheets are raw text files and can be read with any basic text editor. However, some formats are proprietary and cannot be read with a text editor, such as Microsoft Excel files (```.xls```).

\* Most import functions assume that the first row of a spreadsheet file is a header with column names. To know if the file has a header, it helps to look at the file with a text editor before trying to import it.

## Paths and the Working Directory

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html#paths-and-the-working-directory){target="\_blank"}.

\*\*Key points\*\*

\* The working directory is where R looks for files and saves files by default.

\* See your working directory with ```getwd()```. Change your working directory with ```setwd()```.

\* We suggest you create a directory for each project and keep your raw data inside that directory.

\* Use the ```file.path()``` function to generate a full path from a relative path and a file name. Use ```file.path()``` instead of ```paste()``` because ```file.path()``` is aware of your operating system and will use the correct slashes to navigate your machine.

\* The ```file.copy()``` function copies a file to a new path.

\*Code\*

```{r, eval=FALSE, echo=TRUE}

# see working directory

getwd()

# change your working directory

setwd()

````

```{r}

# set path to the location for raw data files in the dslabs package and list files

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

list.files(path)

# generate a full path to a file

filename <- "murders.csv"

fullpath <- file.path(path, filename)

fullpath

# copy file from dslabs package to your working directory

file.copy(fullpath, getwd())

# check if the file exists

file.exists(filename)

```

## The readr and readxl Packages

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html#the-readr-and-readxl-packages){target="\_blank"}.

\*\*Key points\*\*

\* \*\*readr\*\* is the \*\*tidyverse\*\* library that includes functions for reading data stored in text file spreadsheets into R. Functions in the package include ```read\_csv()```, ```read\_tsv()```, ```read\_delim()``` and more. These differ by the delimiter they use to split columns.

\* The \*\*readxl\*\* package provides functions to read Microsoft Excel formatted files.

\* The ```excel\_sheets()``` function gives the names of the sheets in the Excel file. These names are passed to the sheet argument for the \*\*readxl\*\* functions ```read\_excel()```, ```read\_xls()``` and ```read\_xlsx()```.

\* The ```read\_lines()``` function shows the first few lines of a file in R.

\*Code\*

```{r}

if(!require(dslabs)) install.packages("dslabs")

if(!require(tidyverse)) install.packages("tidyverse")

if(!require(readxl)) install.packages("readxl")

library(dslabs)

library(tidyverse) # includes readr

library(readxl)

# inspect the first 3 lines

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

# read file in CSV format

dat <- read\_csv(filename)

#read using full path

dat <- read\_csv(fullpath)

head(dat)

#Ex：

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

files <- list.files(path)

files

filename <- "murders.csv"

filename1 <- "life-expectancy-and-fertility-two-countries-example.csv"

filename2 <- "fertility-two-countries-example.csv"

dat=read.csv(file.path(path, filename))

dat1=read.csv(file.path(path, filename1))

dat2=read.csv(file.path(path, filename2))

```

## Importing Data Using R-base Functions

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html#r-base-importing-functions){target="\_blank"}.

\*\*Key point\*\*

\* R-base import functions (```read.csv()```, ```read.table()```, ```read.delim()```) generate data frames rather than tibbles and character variables are converted to factors. This can be avoided by setting the argument ```stringsAsFactors=FALSE```.

\*Code\*

```{r}

# read.csv converts strings to factors

dat2 <- read.csv(filename)

class(dat2$abb)

class(dat2$region)

```

## Downloading Files from the Internet

The textbook for this section is available [here](https://rafalab.github.io/dsbook/importing-data.html#downloading-files){target="\_blank"}.

\*\*Key points\*\*

\* The ```read\_csv()``` function and other import functions can read a URL directly.

\* If you want to have a local copy of the file, you can use ```download.file()```.

\* ```tempdir()``` creates a directory with a name that is very unlikely not to be unique.

\* ```tempfile()``` creates a character string that is likely to be a unique filename.

\*Code\*

```{r}

url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"

dat <- read\_csv(url)

download.file(url, "murders.csv")

tempfile()

tmp\_filename <- tempfile()

download.file(url, tmp\_filename)

dat <- read\_csv(tmp\_filename)

file.remove(tmp\_filename)

```

## Assessment Part 1 - Data Import

1. Which of the following is NOT part of the data wrangling process?

- [ ] A. Importing data into R

- [ ] B. Formatting dates/times

- [X] C. Checking correlations between your variables

- [ ] D. Tidying data

2. Which files could be opened in a basic text editor?

Select ALL that apply.

- [X] A. data.txt

- [X] B. data.csv

- [ ] C. data.xlsx

- [X] D. data.tsv

3. You want to analyze a file containing race finish times for a recent marathon. You open the file in a basic text editor and see lines that look like the following:

```{r, eval=FALSE, echo=TRUE}

initials,state,age,time

vib,MA,61,6:01

adc,TX,45,5:45

kme,CT,50,4:19

```

What type of file is this?

- [ ] A. A comma-delimited file without a header

- [ ] B. A tab-delimited file with a header

- [ ] C. A white space-delimited file without a header

- [X] D. A comma-delimited file with a header

4. Assume the following is the full path to the directory that a student wants to use as their working directory in R: “/Users/student/Documents/projects/”

Which of the following lines of code CANNOT set the working directory to the desired “projects” directory?

- [ ] A. ```setwd("~/Documents/projects/")```

- [ ] B. ```setwd("/Users/student/Documents/projects/")```

- [X] C. ```setwd(/Users/student/Documents/projects/)```

- [ ] D. ```dir <- "/Users/student/Documents/projects" setwd(dir)```

5. We want to copy the “murders.csv” file from the dslabs package into an existing folder “data”, which is located in our HarvardX-Wrangling projects folder. We first enter the code below into our RStudio console.

```{r, eval=FALSE, echo=TRUE}

> getwd()

[1] "C:/Users/UNIVERSITY/Documents/Analyses/HarvardX-Wrangling"

> filename <- "murders.csv"

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

```

Which of the following commands would NOT successfully copy “murders.csv” into the folder “data”?

- [X] A.

```file.copy(file.path(path, "murders.csv"), getwd())```

- [ ] B.

```setwd("data")

file.copy(file.path(path, filename), getwd())

```

- [ ] C.

```file.copy(file.path(path, "murders.csv"), file.path(getwd(), "data"))```

- [ ] D.

```

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

file.destination <- file.path(getwd(),"data")

file.copy(file.location, file.destination)

```

6. You are not sure whether the murders.csv file has a header row. How could you check this?

Select ALL that apply.

- [X] A. Open the file in a basic text editor.

- [X] B. In the RStudio “Files” pane, click on your file, then select “View File”.

- [X] C. Use the command ```read\_lines``` (remembering to specify the number of rows with the ```n\_max``` argument).

7. What is one difference between ```read\_excel``` and ```read\_xlsx```?

- [ ] A. ```read\_excel()``` also reads meta-data from the excel file, such as sheet names, while ```read\_xlsx()``` only reads the first sheet in a file.

- [X] B. ```read\_excel()``` reads both .xls and .xlsx files by detecting the file format from its extension, while ```read\_xlsx()``` only reads .xlsx files.

- [ ] C. ```read\_excel()``` is part of the \*\*readr\*\* package, while ```read\_xlsx()``` is part of the \*\*readxl\*\* package and has more options.

- [ ] D. ```read\_xlsx()``` has been replaced by ```read\_excel()``` in a recent \*\*readxl\*\* package update.

8. You have a file called “times.txt” that contains race finish times for a marathon. The first four lines of the file look like this:

```{r, eval=FALSE, echo=TRUE}

initials,state,age,time

vib,MA,61,6:01

adc,TX,45,5:45

kme,CT,50,4:19

```

Which line of code will NOT produce a tibble with column names “initials”, “state”, “age”, and “time”?

- [ ] A. ```race\_times <- read\_csv("times.txt")```

- [X] B. ```race\_times <- read.csv("times.txt")```

- [ ] C. ```race\_times <- read\_csv("times.txt", col\_names = TRUE)```

- [ ] D. ```race\_times <- read\_delim("times.txt", delim = “,”)```

9. You also have access to marathon finish times in the form of an Excel document named “times.xlsx”. In the Excel document, different sheets contain race information for different years. The first sheet is named “2015”, the second is named “2016”, and the third is named “2017”.

Which line of code will NOT import the data contained in the “2016” tab of this Excel sheet?

- [ ] A. ```times\_2016 <- read\_excel("times.xlsx", sheet = 2)```

- [X] B. ```times\_2016 <- read\_xlsx("times.xlsx", sheet = “2”)```

- [ ] C. ```times\_2016 <- read\_excel("times.xlsx", sheet = "2016")```

- [ ] D. ```times\_2016 <- read\_xlsx("times.xlsx", sheet = 2)```

10. You have a comma-separated values file that contains the initials, home states, ages, and race finish times for marathon runners. The runners’ initials contain three characters for the runners’ first, middle, and last names (for example, “KME”).

You read in the file using the following code.

```{r, eval=FALSE, echo=TRUE}

race\_times <- read.csv(“times.csv”)

```

What is the data type of the initials in the object ```race\_times```?

- [ ] A. integers

- [ ] B. characters

- [X] C. factors

- [ ] D. logical

11. Which of the following is NOT a real difference between the readr import functions and the base R import functions?

- [ ] A. The import functions in the readr package all start as ```read\_```, while the import functions for base R all start with ```read```.

- [ ] B. Base R import functions automatically convert character columns to factors.

- [X] C. The base R import functions can read .csv files, but cannot files with other delimiters, such as .tsv files, or fixed-width files.

- [ ] D. Base R functions import data as a data frame, while readr functions import data as a tibble.

12. You read in a file containing runner information and marathon finish times using the following code.

```{r, eval=FALSE, echo=TRUE}

race\_times <- read.csv(“times.csv”, stringsAsFactors = F)

```

What is the class of the object ```race\_times```?

- [X] A. data frame

- [ ] B. tibble

- [ ] C. matrix

- [ ] D. vector

13. Select the answer choice that summarizes all of the actions that the following lines of code can perform. Please note that the url below is an example and does not lead to data.

```{r, eval=FALSE, echo=TRUE}

url <- "https://raw.githubusercontent.com/MyUserName/MyProject/master/MyData.csv "

dat <- read\_csv(url)

download.file(url, "MyData.csv")

```

- [ ] A. Create a tibble in R called ```dat``` that contains the information contained in the csv file stored on Github and save that tibble to the working directory.

- [ ] B. Create a matrix in R called ```dat``` that contains the information contained in the csv file stored on Github. Download the csv file to the working directory and name the downloaded file “MyData.csv”.

- [ ] C. Create a tibble in R called ```dat``` that contains the information contained in the csv file stored on Github. Download the csv file to the working directory and randomly assign it a temporary name that is very likely to be unique.

- [X] D. Create a tibble in R called ```dat``` that contains the information contained in the csv file stored on Github. Download the csv file to the working directory and name the downloaded file “MyData.csv”.

## Assessment Part 2 - Data Import

14. Inspect the file at the following URL:

https://raw.githubusercontent.com/rasbt/python-machine-learning-book/master/code/datasets/wdbc/wdbc.data

Which \*\*readr\*\* function should be used to import this file?

- [ ] A. ```read\_table()```

- [X] B. ```read\_csv()```

- [ ] C. ```read\_csv2()```

- [ ] D. ```read\_tsv()```

- [ ] E. None of the above

15. Check the documentation for the readr function you chose in the previous question to learn about its arguments. Determine which arguments you need to the file from the previous question:

```{r, eval=FALSE, echo=TRUE}

url <- "https://raw.githubusercontent.com/rasbt/python-machine-learning-book/master/code/datasets/wdbc/wdbc.data"

```

Does this file have a header row? Does the \*\*readr\*\* function you chose need any additional arguments to import the data correctly?

- [ ] A. Yes, there is a header. No arguments are needed.

- [ ] B. Yes, there is a header. The ```header=TRUE``` argument is necessary.

- [ ] C. Yes, there is a header. The ```col\_names=TRUE``` argument is necessary.

- [ ] D. No, there is no header. No arguments are needed.

- [ ] E. No, there is no header. The ```header=FALSE``` argument is necessary.

- [X] F. No, there is no header. The ```col\_names=FALSE``` argument is necessary.

16. Inspect the imported data from the previous question.

How many rows are in the dataset?

```{r}

url <- "https://raw.githubusercontent.com/rasbt/python-machine-learning-book/master/code/datasets/wdbc/wdbc.data"

df <- read\_csv(url, col\_names = FALSE)

nrow(df)

```

How many columns are in the dataset?

```{r}

ncol(df)

```

# Section 2 Overview

In the \*\*Tidy Data\*\* section, you will learn how to convert data from a raw to a tidy format.

This section is divided into three parts: \*\*Reshaping Data\*\*, \*\*Combining Tables\*\*, and \*\*Web Scraping\*\*.

After completing the \*\*Tidy Data\*\* section, you will be able to:

\* \*\*Reshape data\*\* using functions from the \*\*tidyr\*\* package, including ```gather()```, ```spread()```, ```separate()```, and ```unite()```.

\* Combine information from different tables using \*\*join\*\* functions from the \*\*dplyr\*\* package.

\* Combine information from different tables using \*\*binding\*\* functions from the \*\*dplyr\*\* package.

\* Use \*\*set operators\*\* to combine data frames.

\* Gather data from a website through \*\*web scraping\*\* and use of \*\*CSS selectors\*\*.

## Tidy Data

The textbook for this section is available [here](https://rafalab.github.io/dsbook/tidyverse.html#tidy-data){target="\_blank"}.

\*\*Key points\*\*

\* In tidy data, each row represents an observation and each column represents a different variable.

\* In wide data, each row includes several observations and one of the variables is stored in the header.

\*Code\*

```{r}

data(gapminder)

# create and inspect a tidy data frame

tidy\_data <- gapminder %>%

filter(country %in% c("South Korea", "Germany")) %>%

select(country, year, fertility)

head(tidy\_data)

# plotting tidy data is simple

tidy\_data %>%

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

geom\_point()

# import and inspect example of original Gapminder data in wide format

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

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

wide\_data <- read\_csv(filename)

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

```

## Reshaping Data

The textbook for this section is available [here](https://rafalab.github.io/dsbook/reshaping-data.html){target="\_blank"}, [here](https://rafalab.github.io/dsbook/reshaping-data.html#gather){target="\_blank"} and [here](https://rafalab.github.io/dsbook/reshaping-data.html#spread){target="\_blank"}.

\*\*Key points\*\*

\* The \*\*tidyr\*\* package includes several functions that are useful for tidying data.

\* The ```gather()``` function converts wide data into tidy data.

\* The ```spread()``` function converts tidy data to wide data.

\*Code\*

```{r}

# original wide data

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

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

wide\_data <- read\_csv(filename)

# tidy data from dslabs

tidy\_data <- gapminder %>%

filter(country %in% c("South Korea", "Germany")) %>%

select(country, year, fertility)

# gather wide data to make new tidy data

new\_tidy\_data <- wide\_data %>%

gather(year, fertility, `1960`:`2015`)

head(new\_tidy\_data)

# gather all columns except country

new\_tidy\_data <- wide\_data %>%

gather(year, fertility, -country)

# gather treats column names as characters by default

class(tidy\_data$year)

class(new\_tidy\_data$year)

# convert gathered column names to numeric

new\_tidy\_data <- wide\_data %>%

gather(year, fertility, -country, convert = TRUE)

class(new\_tidy\_data$year)

# ggplot works on new tidy data

new\_tidy\_data %>%

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

geom\_point()

# spread tidy data to generate wide data

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

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

```

## Separate and Unite

The textbook for this section is available [here](https://rafalab.github.io/dsbook/reshaping-data.html#separate){target="\_blank"} and [here](https://rafalab.github.io/dsbook/reshaping-data.html#unite){target="\_blank"}.

\*\*Key points\*\*

\* The ```separate()``` function splits one column into two or more columns at a specified character that separates the variables.

\* When there is an extra separation in some of the entries, use ```fill="right"``` to pad missing values with NAs, or use ```extra="merge"``` to keep extra elements together.

\* The ```unite()``` function combines two columns and adds a separating character.

\*Code\*

```{r}

# import data

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

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

raw\_dat <- read\_csv(filename)

select(raw\_dat, 1:5)

# gather all columns except country

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

head(dat)

dat$key[1:5]

# separate on underscores

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

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

# split on all underscores, pad empty cells with NA

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

fill = "right")

# split on first underscore but keep life\_expectancy merged

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

# separate then spread

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

spread(variable\_name, value)

# separate then unite

dat %>%

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

unite(variable\_name, first\_variable\_name, second\_variable\_name, sep="\_")

# full code for tidying data

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)

```

## Assessment Part 1 - Reshaping Data

1. A collaborator sends you a file containing data for three years of average race finish times.

```{r, eval=FALSE, echo=TRUE}

age\_group,2015,2016,2017

20,3:46,3:22,3:50

30,3:50,3:43,4:43

40,4:39,3:49,4:51

50,4:48,4:59,5:01

```

Are these data considered “tidy” in R? Why or why not?

- [ ] A. Yes. These data are considered “tidy” because each row contains unique observations.

- [ ] B. Yes. These data are considered “tidy” because there are no missing data in the data frame.

- [X] C. No. These data are not considered “tidy” because the variable “year” is stored in the header.

- [ ] D. No. These data are not considered “tidy” because there are not an equal number of columns and rows.

2. Below are four versions of the same dataset. Which one is in a tidy format?

- [X] A.

```{r, eval=FALSE, echo=TRUE}

state abb region population total

Alabama AL South 4779736 135

Alaska AK West 710231 19

Arizona AZ West 6392017 232

Arkansas AR South 2915918 93

California CA West 37253956 1257

Colorado CO West 5029196 65

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

state abb region var people

Alabama AL South population 4779736

Alabama AL South total 135

Alaska AK West population 710231

Alaska AK West total 19

Arizona AZ West population 6392017

Arizona AZ West total 232

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

state abb Northeast South North Central West

Alabama AL NA 4779736 NA NA

Alaska AK NA NA NA 710231

Arizona AZ NA NA NA 6392017

Arkansas AR NA 2915918 NA NA

California CA NA NA NA 37253956

Colorado CO NA NA NA 5029196

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

state abb region rate

Alabama AL South 2.82e-05

Alaska AK West 2.68e-05

Arizona AZ West 3.63e-05

Arkansas AR South 3.19e-05

California CA West 3.37e-05

Colorado CO West 1.29e-05

```

3. Your file called “times.csv” has age groups and average race finish times for three years of marathons.

```{r, eval=FALSE, echo=TRUE}

age\_group,2015,2016,2017

20,3:46,3:22,3:50

30,3:50,3:43,4:43

40,4:39,3:49,4:51

50,4:48,4:59,5:01

```

You read in the data file using the following command.

```{r}

d <- read\_csv("files/times.csv")

```

Which commands will help you “tidy” the data?

```{r}

tidy\_data <- d %>%

gather(year, time, `2015`:`2017`)

tidy\_data

```

- [X] A.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(year, time, `2015`:`2017`)

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

spread(year, time, `2015`:`2017`)

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(age\_group, year, time, `2015`:`2017`)

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(time, `2015`:`2017`)

```

4. You have a dataset on U.S. contagious diseases, but it is in the following wide format:

```{r, eval=FALSE, echo=TRUE}

> head(dat\_wide)

state year population Hepatitis A Mumps Polio Rubella

Alabama 1990 4040587 86 19 76 1

Alabama 1991 4066003 39 14 65 0

Alabama 1992 4097169 35 12 24 0

Alabama 1993 4133242 40 22 67 0

Alabama 1994 4173361 72 12 39 0

Alabama 1995 4216645 75 2 38 0

```

Which of the following would transform this into a tidy dataset, with each row representing an observation of the incidence of each specific disease (as shown below)?

```{r, eval=FALSE, echo=TRUE}

> head(dat\_tidy)

state year population disease count

Alabama 1990 4040587 Hepatitis A 86

Alabama 1991 4066003 Hepatitis A 39

Alabama 1992 4097169 Hepatitis A 35

Alabama 1993 4133242 Hepatitis A 40

Alabama 1994 4173361 Hepatitis A 72

Alabama 1995 4216645 Hepatitis A 75

```

- [ ] A.

```{r, eval=FALSE, echo=TRUE}

dat\_tidy <- dat\_wide %>%

gather (key = count, value = disease, `Hepatitis A`, `Rubella`)

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

dat\_tidy <- dat\_wide %>%

gather(key = count, value = disease, -state, -year, -population)

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

dat\_tidy <- dat\_wide %>%

gather(key = disease, value = count, -state)

```

- [X] D.

```{r, eval=FALSE, echo=TRUE}

dat\_tidy <- dat\_wide %>%

gather(key = disease, value = count, “Hepatitis A”: “Rubella”)

```

5. You have successfully formatted marathon finish times into a tidy object called ```tidy\_data```. The first few lines are shown below.

```{r, eval=FALSE, echo=TRUE}

age\_group year time

20 2015 03:46

30 2015 03:50

40 2015 04:39

50 2015 04:48

20 2016 03:22

```

Select the code that converts these data back to the wide format, where each year has a separate column.

```{r}

tidy\_data %>% spread(year, time)

```

- [ ] A. ```tidy\_data %>% spread(time, year)```

- [X] B. ```tidy\_data %>% spread(year, time)```

- [ ] C. ```tidy\_data %>% spread(year, age\_group)```

- [ ] D. ```tidy\_data %>% spread(time, year, `2015`:`2017`)```

6. You have the following dataset:

```{r, eval=FALSE, echo=TRUE}

> head(dat)

state abb region var people

Alabama AL South population 4779736

Alabama AL South total 135

Alaska AK West population 710231

Alaska AK West total 19

Arizona AZ West population 6392017

Arizona AZ West total 232

```

You would like to transform it into a dataset where population and total are each their own column (shown below). Which code would best accomplish this?

```{r, eval=FALSE, echo=TRUE}

state abb region population total

Alabama AL South 4779736 135

Alaska AK West 710231 19

Arizona AZ West 6392017 232

Arkansas AR South 2915918 93

California CA West 37253956 1257

Colorado CO West 5029196 65

```

- [X] A. ```dat\_tidy <- dat %>% spread(key = var, value = people)```

- [ ] B. ```dat\_tidy <- dat %>% spread(key = state:region, value = people)```

- [ ] C. ```dat\_tidy <- dat %>% spread(key = people, value = var)```

- [ ] D. ```dat\_tidy <- dat %>% spread(key = region, value = people)```

7. A collaborator sends you a file containing data for two years of average race finish times, "times2.csv":.

```{r, eval=FALSE, echo=TRUE}

age\_group,2015\_time,2015\_participants,2016\_time,2016\_participants

20,3:46,54,3:22,62

30,3:50,60,3:43,58

40,4:39,29,3:49,33

50,4:48,10,4:59,14

```

You read in the data file

```{r}

d <- read\_csv("files/times2.csv")

```

Which of the answers below best tidys the data?

```{r}

tidy\_data <- d %>%

gather(key = "key", value = "value", -age\_group) %>%

separate(col = key, into = c("year", "variable\_name"), sep = "\_") %>%

spread(key = variable\_name, value = value)

tidy\_data

```

- [ ] A.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(key = “key”, value = “value”, -age\_group) %>%

separate(col = key, into = c(“year”, “variable\_name”), sep = “.”) %>%

spread(key = variable\_name, value = value)

```

- [X] B.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(key = “key”, value = “value”, -age\_group) %>%

separate(col = key, into = c(“year”, “variable\_name”), sep = “\_”) %>%

spread(key = variable\_name, value = value)

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(key = “key”, value = “value”) %>%

separate(col = key, into = c(“year”, “variable\_name”), sep = “\_”) %>%

spread(key = variable\_name, value = value)

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- d %>%

gather(key = “key”, value = “value”, -age\_group) %>%

separate(col = key, into = “year”, sep = “\_”) %>%

spread(key = year, value = value)

```

8. You are in the process of tidying some data on heights, hand length, and wingspan for basketball players in the draft. Currently, you have the following:

```{r, eval=FALSE, echo=TRUE}

> head(stats)

key value

allen\_height 75

allen\_hand\_length 8.25

allen\_wingspan 79.25

bamba\_height 83.25

bamba\_hand\_length 9.75

bamba\_wingspan 94

```

Select all of the correct commands below that would turn this data into a “tidy” format.

- [X] A.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- stats %>%

separate(col = key, into = c("player", "variable\_name"), sep = "\_", extra = "merge") %>%

spread(key = variable\_name, value = value)

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- stats %>%

separate(col = key, into = c("player", "variable\_name1", "variable\_name2"), sep = "\_", fill = "right") %>%

unite(col = variable\_name, variable\_name1, variable\_name2, sep = "\_") %>%

spread(key = variable\_name, value = value)

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

tidy\_data <- stats %>%

separate(col = key, into = c("player", "variable\_name"), sep = "\_") %>%

spread(key = variable\_name, value = value)

```

## Assessment Part 2 - Reshaping Data

9. Examine the built-in dataset ```co2```. This dataset comes with base R, not \*\*dslabs\*\* - just type ```co2``` to access the dataset.

```{r}

co2

```

Is ```co2``` tidy? Why or why not?

- [ ] A. ```co2``` is tidy data: it has one year for each row.

- [ ] B. ```co2``` is tidy data: each column is a different month.

- [ ] C. ```co2``` is not tidy: there are multiple observations per column.

- [X] D. ```co2``` is not tidy: to be tidy we would have to wrangle it to have three columns (year, month and value), and then each co2 observation would have a row.

10. Run the following command to define the co2\_wide object:

```{r}

co2\_wide <- data.frame(matrix(co2, ncol = 12, byrow = TRUE)) %>%

setNames(1:12) %>%

mutate(year = as.character(1959:1997))

```

Use the ```gather()``` function to make this dataset tidy. Call the column with the CO2 measurements ```co2``` and call the month column ```month```. Name the resulting object ```co2\_tidy```.

Which code would return the correct tidy format?

```{r}

co2\_tidy <- gather(co2\_wide,month,co2,-year)

```

- [ ] A. ```co2\_tidy <- gather(co2\_wide,month,co2,year)```

- [ ] B. ```co2\_tidy <- gather(co2\_wide,co2,month,-year)```

- [ ] C. ```co2\_tidy <- gather(co2\_wide,co2,month,year)```

- [X] D. ```co2\_tidy <- gather(co2\_wide,month,co2,-year)```

11. Use ```co2\_tidy``` to plot CO2 versus month with a different curve for each year:

```{r}

co2\_tidy %>% ggplot(aes(as.numeric(month), co2, color = year)) + geom\_line()

```

What can be concluded from this plot?

- [ ] A. CO2 concentrations increased monotonically (never decreased) from 1959 to 1997.

- [X] B. CO2 concentrations are highest around May and the yearly average increased from 1959 to 1997.

- [ ] C. CO2 concentrations are highest around October and the yearly average increased from 1959 to 1997.

- [ ] D. Yearly average CO2 concentrations have remained constant over time.

- [ ] E. CO2 concentrations do not have a seasonal trend.

12. Load the ```admissions``` dataset from \*\*dslabs\*\*, which contains college admission information for men and women across six majors, and remove the ```applicants``` percentage column:

```{r}

data(admissions)

dat <- admissions %>% select(-applicants)

```

Your goal is to get the data in the shape that has one row for each major, like this:

```{r, eval=FALSE, echo=TRUE}

major men women

A 62 82

B 63 68

C 37 34

D 33 35

E 28 24

F 6 7

```

Which command could help you to wrangle the data into the desired format?

```{r}

dat\_tidy <- spread(dat, gender, admitted)

```

- [ ] A. ```dat\_tidy <- spread(dat, major, admitted)```

- [ ] B. ```dat\_tidy <- spread(dat, gender, major)```

- [X] C. ```dat\_tidy <- spread(dat, gender, admitted)```

- [ ] D. ```dat\_tidy <- spread(dat, admitted, gender)```

13. Now use the ```admissions``` dataset to create the object ```tmp```, which has columns ```major```, ```gender```, ```key``` and ```value```:

```{r}

tmp <- gather(admissions, key, value, admitted:applicants)

tmp

```

Combine the key and gender and create a new column called ```column\_name``` to get a variable with the following values: ```admitted\_men```, ```admitted\_women```, ```applicants\_men ```and ```applicants\_women```. Save the new data as ```tmp2```.

Which command could help you to wrangle the data into the desired format?

```{r}

tmp2 <- unite(tmp, column\_name, c(key, gender))

```

- [ ] A. ```tmp2 <- spread(tmp, column\_name, key, gender)```

- [ ] B. ```tmp2 <- gather(tmp, column\_name, c(gender,key))```

- [ ] C. ```tmp2 <- unite(tmp, column\_name, c(gender, key))```

- [ ] D. ```tmp2 <- spread(tmp, column\_name, c(key,gender))```

- [X] E. ```tmp2 <- unite(tmp, column\_name, c(key, gender))```

14. Which function can reshape ```tmp2``` to a table with six rows and five columns named ```major```, ```admitted\_men```, ```admitted\_women```, ```applicants\_men``` and ```applicants\_women```?

```{r}

spread(tmp2, column\_name, value)

```

- [ ] A. ```gather()```

- [X] B. ```spread()```

- [ ] C. ```separate()```

- [ ] D. ```unite()```

## Combining Tables

The textbook for this section is available [here](https://rafalab.github.io/dsbook/joining-tables.html){target="\_blank"}.

\*\*Key points\*\*

\* The join functions in the \*\*dplyr\*\* package combine two tables such that matching rows are together.

\* ```left\_join()``` only keeps rows that have information in the first table.

\* ```right\_join()``` only keeps rows that have information in the second table.

\* ```inner\_join()``` only keeps rows that have information in both tables.

\* ```full\_join()``` keeps all rows from both tables.

\* ```semi\_join()``` keeps the part of first table for which we have information in the second.

\* ```anti\_join()``` keeps the elements of the first table for which there is no information in the second.

\*Code\*

```{r}

if(!require(ggrepel)) install.packages("ggrepel")

# import US murders data

library(ggrepel)

ds\_theme\_set()

data(murders)

head(murders)

# import US election results data

data(polls\_us\_election\_2016)

head(results\_us\_election\_2016)

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

# join the murders table and US election results table

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

head(tab)

# plot electoral votes versus population

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)

# make two smaller tables to demonstrate joins

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

tab1

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

tab2

# experiment with different joins

left\_join(tab1, tab2)

tab1 %>% left\_join(tab2)

tab1 %>% right\_join(tab2)

inner\_join(tab1, tab2)

semi\_join(tab1, tab2)

anti\_join(tab1, tab2)

```

## Binding

The textbook for this section is available [here](https://rafalab.github.io/dsbook/joining-tables.html#binding){target="\_blank"}.

\*\*Key points\*\*

\* Unlike the join functions, the binding functions do not try to match by a variable, but rather just combine datasets.

\* ```bind\_cols()``` binds two objects by making them columns in a tibble. The R-base function ```cbind()``` binds columns but makes a data frame or matrix instead.

\* The ```bind\_rows()``` function is similar but binds rows instead of columns. The R-base function ```rbind()``` binds rows but makes a data frame or matrix instead.

\*Code\*

```{r}

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

tab1 <- tab[, 1:3]

tab2 <- tab[, 4:6]

tab3 <- tab[, 7:9]

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

head(new\_tab)

tab1 <- tab[1:2,]

tab2 <- tab[3:4,]

bind\_rows(tab1, tab2)

```

## Set Operators

The textbook for this section is available [here](https://rafalab.github.io/dsbook/joining-tables.html#set-operators){target="\_blank"}.

\*\*Key points\*\*

\* By default, the set operators in R-base work on vectors. If \*\*tidyverse/dplyr\*\* are loaded, they also work on data frames.

\* You can take intersections of vectors using ```intersect()```. This returns the elements common to both sets.

\* You can take the union of vectors using ```union()```. This returns the elements that are in either set.

\* The set difference between a first and second argument can be obtained with ```setdiff()```. Note that this function is not symmetric.

\* The function ```set\_equal()``` tells us if two sets are the same, regardless of the order of elements.

\*Code\*

```{r}

# intersect vectors or data frames

intersect(1:10, 6:15)

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

tab1 <- tab[1:5,]

tab2 <- tab[3:7,]

intersect(tab1, tab2)

# perform a union of vectors or data frames

union(1:10, 6:15)

union(c("a","b","c"), c("b","c","d"))

tab1 <- tab[1:5,]

tab2 <- tab[3:7,]

union(tab1, tab2)

# set difference of vectors or data frames

setdiff(1:10, 6:15)

setdiff(6:15, 1:10)

tab1 <- tab[1:5,]

tab2 <- tab[3:7,]

setdiff(tab1, tab2)

# setequal determines whether sets have the same elements, regardless of order

setequal(1:5, 1:6)

setequal(1:5, 5:1)

setequal(tab1, tab2)

```

## Assessment - Combining Tables

1. You have created a ```tab1``` and ```tab2``` of state population and election data:

```{r, eval=FALSE, echo=TRUE}

> tab1

state population

Alabama 4779736

Alaska 710231

Arizona 6392017

Delaware 897934

District of Columbia 601723

> tab2

state electoral\_votes

Alabama 9

Alaska 3

Arizona 11

California 55

Colorado 9

Connecticut 7

> dim(tab1)

[1] 5 2

> dim(tab2)

[1] 6 2

```

What are the dimensions of the table dat, created by the following command?

```{r, eval=FALSE, echo=TRUE}

dat <- left\_join(tab1, tab2, by = “state”)

```

- [ ] A. 3 rows by 3 columns

- [ ] B. 5 rows by 2 columns

- [X] C. 5 rows by 3 columns

- [ ] D. 6 rows by 3 columns

2. We are still using the ```tab1``` and ```tab2 tables shown in question 1. What join command would create a new table “dat” with three rows and two columns?

- [ ] A. ```dat <- right\_join(tab1, tab2, by = “state”)```

- [ ] B. ```dat <- full\_join(tab1, tab2, by = “state”)```

- [ ] C. ```dat <- inner\_join(tab1, tab2, by = “state”)```

- [X] D. ```dat <- semi\_join(tab1, tab2, by = “state”)```

3. Which of the following are real differences between the join and bind functions?

- [X] A. Binding functions combine by position, while join functions match by variables.

- [X] B. Joining functions can join datasets of different dimensions, but the bind functions must match on the appropriate dimension (either same row or column numbers).

- [X] C. Bind functions can combine both vectors and dataframes, while join functions work for only for dataframes.

- [ ] D. The join functions are a part of the dplyr package and have been optimized for speed, while the bind functions are inefficient base functions.

4. We have two simple tables, shown below, with columns ```x``` and ```y```:

```{r, eval=FALSE, echo=TRUE}

> df1

x y

a a

b a

> df2

x y

a a

a b

```

Which command would result in the following table?

```{r, eval=FALSE, echo=TRUE}

> final

x y

b a

```

- [ ] A. ```final <- union(df1, df2)```

- [X] B. ```final <- setdiff(df1, df2)```

- [ ] C. ```final <- setdiff(df2, df1)```

- [ ] D. ```final <- intersect(df1, df2)```

Install and load the \*\*Lahman\*\* library. This library contains a variety of datasets related to US professional baseball. We will use this library for the next few questions and will discuss it more extensively in the Regression course. For now, focus on wrangling the data rather than understanding the statistics.

The ```Batting``` data frame contains the offensive statistics for all baseball players over several seasons. Filter this data frame to define ```top``` as the top 10 home run (```HR```) hitters in 2016:

```{r}

if(!require(Lahman)) install.packages("Lahman")

library(Lahman)

top <- Batting %>%

filter(yearID == 2016) %>%

arrange(desc(HR)) %>% # arrange by descending HR count

slice(1:10) # take entries 1-10

top %>% as\_tibble()

```

Also Inspect the ```Master``` data frame, which has demographic information for all players:

```{r}

Master %>% as\_tibble()

```

5. Use the correct ```join``` or ```bind``` function to create a combined table of the names and statistics of the top 10 home run (HR) hitters for 2016. This table should have the player ID, first name, last name, and number of HR for the top 10 players. Name this data frame ```top\_names```.

Identify the ```join``` or ```bind``` that fills the blank in this code to create the correct table:

```{r, eval=FALSE, echo=TRUE}

top\_names <- top %>% \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ %>%

select(playerID, nameFirst, nameLast, HR)

```

Which bind or join function fills the blank to generate the correct table?

```{r}

top\_names <- top %>% left\_join(Master) %>%

select(playerID, nameFirst, nameLast, HR)

```

- [ ] A. ```rbind(Master)```

- [ ] B. ```cbind(Master)```

- [X] C. ```left\_join(Master)```

- [ ] D. ```right\_join(Master)```

- [ ] E. ```full\_join(Master)```

- [ ] F. ```anti\_join(Master)```

6. Inspect the ```Salaries``` data frame. Filter this data frame to the 2016 salaries, then use the correct bind join function to add a ```salary``` column to the ```top\_names``` data frame from the previous question. Name the new data frame ```top\_salary```. Use this code framework:

```{r, eval=FALSE, echo=TRUE}

top\_salary <- Salaries %>% filter(yearID == 2016) %>%

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ %>%

select(nameFirst, nameLast, teamID, HR, salary)

```

Which ```bind``` or ```join``` function fills the blank to generate the correct table?

```{r, eval=FALSE, echo=TRUE}

top\_salary <- Salaries %>% filter(yearID == 2016) %>%

right\_join(top\_names) %>%

select(nameFirst, nameLast, teamID, HR, salary)

```

- [ ] A. ```rbind(top\_names)```

- [ ] B. ```cbind(top\_names)```

- [ ] C. ```left\_join(top\_names)```

- [X] D. ```right\_join(top\_names)```

- [ ] E. ```full\_join(top\_names)```

- [ ] F. ```anti\_join(top\_names)```

7. Inspect the ```AwardsPlayers``` table. Filter awards to include only the year 2016.

How many players from the top 10 home run hitters won at least one award in 2016? Use a set operator.

```{r}

Awards\_2016 <- AwardsPlayers %>% filter(yearID == 2016)

length(intersect(Awards\_2016$playerID, top\_names$playerID))

```

How many players won an award in 2016 but were not one of the top 10 home run hitters in 2016? Use a set operator.

```{r}

length(setdiff(Awards\_2016$playerID, top\_names$playerID))

```

## Web Scraping

The textbook for this section is available [here](https://rafalab.github.io/dsbook/web-scraping.html){target="\_blank"} through [section 23.2](https://rafalab.github.io/dsbook/web-scraping.html#the-rvest-package){target="\_blank"}.

\*\*Key points\*\*

\* Web scraping is extracting data from a website.

\* The \*\*rvest\*\* web harvesting package includes functions to extract nodes of an HTML document: ```html\_nodes()``` extracts all nodes of different types, and ```html\_node()``` extracts the first node.

\* ```html\_table()``` converts an HTML table to a data frame.

\*Code\*

```{r}

# import a webpage into R

if(!require(rvest)) install.packages("rvest")

library(rvest)

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

h <- read\_html(url)

class(h)

h

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

tab <- tab[[2]]

tab <- tab %>% html\_table

class(tab)

tab <- tab %>% setNames(c("state", "population", "total", "murders", "gun\_murders", "gun\_ownership", "total\_rate", "murder\_rate", "gun\_murder\_rate"))

head(tab)

```

## CSS Selectors

This page corresponds to the [textbook section on CSS selectors](https://rafalab.github.io/dsbook/web-scraping.html#css-selectors){target="\_blank"}.

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](http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609). Looking at the code for this page, it seems that the task is impossibly complex. However, selector gadgets actually make this possible. [SelectorGadget](https://selectorgadget.com){target="\_blank"} 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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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.

## Assessment - Web Scraping

Load the following web page, which contains information about Major League Baseball payrolls, into R:

[https://web.archive.org/web/20181024132313/http://www.stevetheump.com/Payrolls.htm](https://web.archive.org/web/20181024132313/http://www.stevetheump.com/Payrolls.htm){target="\_blank"}

```{r}

url <- "https://web.archive.org/web/20181024132313/http://www.stevetheump.com/Payrolls.htm"

h <- read\_html(url)

```

We learned that tables in html are associated with the ```table``` node. Use the ```html\_nodes()``` function and the ```table``` node type to extract the first table. Store it in an object nodes:

```{r}

nodes <- html\_nodes(h, "table")

```

The ```html\_nodes()``` function returns a list of objects of class ```xml\_node```. We can see the content of each one using, for example, the ```html\_text()``` function. You can see the content for an arbitrarily picked component like this:

```{r}

html\_text(nodes[[8]])

```

If the content of this object is an html table, we can use the ```html\_table()``` function to convert it to a data frame:

```{r}

html\_table(nodes[[8]])

```

You will analyze the tables from this HTML page over questions 1-3.

1. Many tables on this page are team payroll tables, with columns for rank, team, and one or more money values.

Convert the first four tables in ```nodes``` to data frames and inspect them.

```{r}

sapply(nodes[1:4], html\_table) # 2, 3, 4 give tables with payroll info

```

Which of the first four ```nodes``` are tables of team payroll? Check all correct answers. Look at table content, not column names.

- [ ] A. None of the above

- [ ] B. Table 1

- [X] C. Table 2

- [X] D. Table 3

- [X] E. Table 4

2. For the last 3 components of ```nodes```, which of the following are true? Check all correct answers.

```{r}

html\_table(nodes[[length(nodes)-2]])

html\_table(nodes[[length(nodes)-1]])

html\_table(nodes[[length(nodes)]])

```

- [X] A. All three entries are tables.

- [ ] B. All three entries are tables of payroll per team.

- [X] C. The last entry shows the average across all teams through time, not payroll per team.

- [ ] D. None of the three entries are tables of payroll per team.

3. Create a table called ```tab\_1``` using entry 10 of ```nodes```. Create a table called ```tab\_2``` using entry 19 of ```nodes```.

Note that the column names should be ```c("Team", "Payroll", "Average")```. You can see that these column names are actually in the first data row of each table, and that ```tab\_1``` has an extra first column ```No.``` that should be removed so that the column names for both tables match.

Remove the extra column in ```tab\_1```, remove the first row of each dataset, and change the column names for each table to ```c("Team", "Payroll", "Average")```. Use a ```full\_join()``` by the ```Team``` to combine these two tables.

Note that some students, presumably because of system differences, have noticed that entry 18 instead of entry 19 of ```nodes``` gives them the ```tab\_2``` correctly; be sure to check entry 18 if entry 19 is giving you problems.

How many rows are in the joined data table?

```{r}

tab\_1 <- html\_table(nodes[[10]])

tab\_2 <- html\_table(nodes[[19]])

col\_names <- c("Team", "Payroll", "Average")

tab\_1 <- tab\_1[-1, -1]

tab\_2 <- tab\_2[-1,]

names(tab\_2) <- col\_names

names(tab\_1) <- col\_names

full\_join(tab\_1,tab\_2, by = "Team")

```

4. The Wikipedia page on [opinion polling for the Brexit referendum](https://en.wikipedia.org/w/index.php?title=Opinion\_polling\_for\_the\_United\_Kingdom\_European\_Union\_membership\_referendum&oldid=896735054){target="\_blank"}, in which the United Kingdom voted to leave the European Union in June 2016, contains several tables. One table contains the results of all polls regarding the referendum over 2016.

![Polls regarding the referendum over 2016](images/Polls\_referendum\_2016.png)

Use the rvest library to read the HTML from this Wikipedia page (make sure to copy both lines of the URL):

```{r}

url <- "https://en.wikipedia.org/w/index.php?title=Opinion\_polling\_for\_the\_United\_Kingdom\_European\_Union\_membership\_referendum&oldid=896735054"

```

Assign ```tab``` to be the html nodes of the "table" class.

How many tables are in this Wikipedia page?

```{r}

tab <- read\_html(url) %>% html\_nodes("table")

length(tab)

```

5. Inspect the first several html tables using ```html\_table()``` with the argument ```fill=TRUE``` (you can read about this argument in the documentation). Find the first table that has 9 columns with the first column named "Date(s) conducted".

What is the first table number to have 9 columns where the first column is named "Date(s) conducted"?

```{r}

tab[[5]] %>% html\_table(fill = TRUE) %>% names() # inspect column names

```

# Section 3 Overview

In the \*\*String Processing\*\* section, we use case studies that help demonstrate how string processing is a powerful tool useful for overcoming many data wrangling challenges. You will see how the original \*\*raw\*\* data was processed to create the data frames we have used in courses throughout this series.

This section is divided into three parts.

After completing the \*\*String Processing\*\* section, you will be able to:

\* \*\*Remove\*\* unwanted characters from text.

\* \*\*Extract numeric\*\* values from text.

\* \*\*Find\*\* and \*\*replace\*\* characters.

\* \*\*Extract specific\*\* parts of strings.

\* \*\*Convert\*\* free form text into more uniform formats.

\* \*\*Split strings\*\* into multiple values.

\* Use \*\*regular expressions (regex)\*\* to process strings.

## String Parsing

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html){target="\_blank"}.

\*\*Key points\*\*

\* The most common tasks in string processing include:

\* extracting numbers from strings

\* removing unwanted characters from text

\* finding and replacing characters

\* extracting specific parts of strings

\* converting free form text to more uniform formats

\* splitting strings into multiple values

\* The \*\*stringr\*\* package in the \*\*tidyverse\*\* contains string processing functions that follow a similar naming format (```str\_functionname```) and are compatible with the pipe.

\*Code\*

```{r}

# read in raw murders data from Wikipedia

url <- "https://en.wikipedia.org/w/index.php?title=Gun\_violence\_in\_the\_United\_States\_by\_state&direction=prev&oldid=810166167"

murders\_raw <- read\_html(url) %>%

html\_nodes("table") %>%

html\_table() %>%

.[[1]] %>%

setNames(c("state", "population", "total", "murder\_rate"))

# inspect data and column classes

head(murders\_raw)

class(murders\_raw$population)

class(murders\_raw$total)

```

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

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#how-to-escape-when-defining-strings){target="\_blank"}.

\*\*Key points\*\*

\* Define a string by surrounding text with either single quotes or double quotes.

\* To include a single quote inside a string, use double quotes on the outside. To include a double quote inside a string, use single quotes on the outside.

\* The ```cat()``` function displays a string as it is represented inside R.

\* To include a double quote inside of a string surrounded by double quotes, use the backslash (\) to escape the double quote. Escape a single quote to include it inside of a string defined by single quotes.

\* We will see additional uses of the escape later.

\*Code\*

```{r}

s <- "Hello!" # double quotes define a string

s <- 'Hello!' # single quotes define a string

```

```{r, eval=FALSE, echo=TRUE}

s <- `Hello` # backquotes do not

s <- "10"" # error - unclosed quotes

```

```{r}

s <- '10"' # correct

# cat shows what the string actually looks like inside R

cat(s)

s <- "5'"

cat(s)

```

```{r, eval=FALSE, echo=TRUE}

# to include both single and double quotes in string, escape with \

s <- '5'10"' # error

s <- "5'10"" # error

```

```{r}

# to include both single and double quotes in string, escape with \

s <- '5\'10"' # correct

cat(s)

s <- "5'10\"" # correct

cat(s)

```

## stringr Package

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#stringr){target="\_blank"}.

\*\*Key points\*\*

\* The main types of string processing tasks are detecting, locating, extracting and replacing elements of strings.

\* The \*\*stringr\*\* package from the \*\*tidyverse\*\* includes a variety of string processing functions that begin with ```str\_``` and take the string as the first argument, which makes them compatible with the pipe.

\*Code\*

```{r}

# murders\_raw defined in web scraping section

# direct conversion to numeric fails because of commas

murders\_raw$population[1:3]

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

```

## Case Study 1: US Murders Data

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#case-study-1-us-murders-data){target="\_blank"}.

\*\*Key points\*\*

\* Use the ```str\_detect()``` function to determine whether a string contains a certain pattern.

\* Use the ```str\_replace\_all()``` function to replace all instances of one pattern with another pattern. To remove a pattern, replace with the empty string (```""```).

\* The ```parse\_number()``` function removes punctuation from strings and converts them to numeric.

\* ```mutate\_at()``` performs the same transformation on the specified column numbers.

\*Code\*

```{r}

# murders\_raw was defined in the web scraping section

# detect whether there are commas

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

murders\_raw %>% summarize\_all(funs(commas))

# replace commas with the empty string and convert to numeric

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

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

# parse\_number also removes commas and converts to numeric

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

identical(test\_1, test\_2)

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

murders\_new %>% head

```

## Assessment - String Processing Part 1

1. Which of the following is NOT an application of string parsing?

- [ ] A. Removing unwanted characters from text.

- [ ] B. Extracting numeric values from text.

- [X] C. Formatting numbers and characters so they can easily be displayed in deliverables like papers and presentations.

- [ ] D. Splitting strings into multiple values.

2. Which of the following commands would not give you an error in R?

- [X] A. ```cat(" LeBron James is 6’8\" ")```

- [ ] B. ```cat(' LeBron James is 6'8" ')```

- [ ] C. ```cat(` LeBron James is 6'8" `)```

- [ ] D. ```cat(" LeBron James is 6\’8" ")```

3. Which of the following are advantages of the \*\*stringr\*\* package over string processing functions in base R? Select all that apply.

- [ ] A. Base R functions are rarely used for string processing by data scientists so it’s not worth learning them.

- [X] B. Functions in stringr all start with “str\_”, which makes them easy to look up using autocomplete.

- [X] C. Stringr functions work better with pipes.

- [X] D. The order of arguments is more consistent in stringr functions than in base R.

4. You have a dataframe of monthly sales and profits in R

```{r, eval=FALSE, echo=TRUE}

> head(dat)

# A tibble: 5 x 3

Month Sales Profit

<chr> <chr> <chr>

January $128,568 $16,234

February $109,523 $12,876

March $115,468 $17,920

April $122,274 $15,825

May $117,921 $15,437

```

Which of the following commands could convert the sales and profits columns to numeric? Select all that apply.

- [X] A. ```dat %>% mutate\_at(2:3, parse\_number)```

- [ ] B. ```dat %>% mutate\_at(2:3, as.numeric)```

- [ ] C. ```dat %>% mutate\_all(parse\_number)```

- [X] D. ```dat %>% mutate\_at(2:3, funs(str\_replace\_all(., c("\\$|,"), ""))) %>%mutate\_at(2:3, as.numeric)```

## Case Study 2: Reported Heights

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#case-study-2-self-reported-heights){target="\_blank"}.

\*\*Key points\*\*

\* In the raw heights data, many students did not report their height as the number of inches as requested. There are many entries with real height information but in the wrong format, which we can extract with string processing.

\* When there are both text and numeric entries in a column, the column will be a character vector. Converting this column to numeric will result in NAs for some entries.

\* To correct problematic entries, look for patterns that are shared across large numbers of entries, then define rules that identify those patterns and use these rules to write string processing tasks.

\* Use ```suppressWarnings()``` to hide warning messages for a function.

\*Code\*

```{r}

# load raw heights data and inspect

data(reported\_heights)

class(reported\_heights$height)

# convert to numeric, inspect, count NAs

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

head(x)

sum(is.na(x))

# keep only entries that result in NAs

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

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

head(n=10)

# calculate cutoffs that cover 99.999% of human population

alpha <- 1/10^6

qnorm(1-alpha/2, 69.1, 2.9)

qnorm(alpha/2, 63.7, 2.7)

# keep only entries that either result in NAs or are outside the plausible range of heights

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

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

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

ind

}

# number of problematic entries

problems <- reported\_heights %>%

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

.$height

length(problems)

# 10 examples of x'y or x'y" or x'y\"

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

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

# 10 examples of x.y or x,y

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

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

# 10 examples of entries in cm rather than inches

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

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

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

```

## Regex

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#regular-expressions){target="\_blank"} through [section 24.5.2](https://rafalab.github.io/dsbook/string-processing.html#special-characters){target="\_blank"}.

\*\*Key points\*\*

\* A regular expression (regex) is a way to describe a specific pattern of characters of text. A set of rules has been designed to do this specifically and efficiently.

\* \*\*stringr\*\* functions can take a regex as a pattern.

\* ```str\_detect()``` indicates whether a pattern is present in a string.

\* The main difference between a regex and a regular string is that a regex can include special characters.

\* The | symbol inside a regex means "or".

\* Use ```'\\d'``` to represent digits. The backlash is used to distinguish it from the character ```'d'```. In R, you must use two backslashes for digits in regular expressions; in some other languages, you will only use one backslash for regex special characters.

\* ```str\_view()``` highlights the first occurrence of a pattern, and the ```str\_view\_all()``` function highlights all occurrences of the pattern.

\*Code\*

```{r, eval=FALSE, echo=TRUE}

# detect whether a comma is present

pattern <- ","

str\_detect(murders\_raw$total, pattern)

# show the subset of strings including "cm"

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

# use the "or" symbol inside a regex (|)

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

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

s <- c(yes, no)

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

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

# highlight the first occurrence of a pattern

str\_view(s, pattern)

# highlight all instances of a pattern

str\_view\_all(s, pattern)

```

## Character Classes, Anchors and Quantifiers

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#character-classes){target="\_blank"}, [here](https://rafalab.github.io/dsbook/string-processing.html#anchors){target="\_blank"} and [here](https://rafalab.github.io/dsbook/string-processing.html#quantifiers){target="\_blank"}

\*\*Key points\*\*

\* Define strings to test your regular expressions, including some elements that match and some that do not. This allows you to check for the two types of errors: failing to match and matching incorrectly.

\* Square brackets define character classes: groups of characters that count as matching the pattern. You can use ranges to define character classes, such as ```[0-9]``` for digits and ```[a-zA-Z]``` for all letters.

\* Anchors define patterns that must start or end at specific places. ```^``` and ```$``` represent the beginning and end of the string respectively.

\* Curly braces are quantifiers that state how many times a certain character can be repeated in the pattern. ```\\d{1,2}``` matches exactly 1 or 2 consecutive digits.

\*Code\*

```{r, eval=FALSE, echo=TRUE}

# s was defined in the previous section

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

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

s <- c(yes, no)

pattern <- "\\d"

# [56] means 5 or 6

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

# [4-7] means 4, 5, 6 or 7

yes <- as.character(4:7)

no <- as.character(1:3)

s <- c(yes, no)

str\_detect(s, "[4-7]")

# ^ means start of string, $ means end of string

pattern <- "^\\d$"

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

no <- c("12", "123", " 1", "a4", "b")

s <- c(yes, no)

str\_view(s, pattern)

# curly braces define quantifiers: 1 or 2 digits

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

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

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

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

# combining character class, anchors and quantifier

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

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

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

str\_detect(yes, pattern)

str\_detect(no, pattern)

```

## Search and Replace with Regex

The textbook for this section is available:

\* [searching and replacing with regex](https://rafalab.github.io/dsbook/string-processing.html#search-and-replace-with-regex){target="\_blank"}.

\* [white space](https://rafalab.github.io/dsbook/string-processing.html#white-space-s){target="\_blank"}.

\* [quantifiers: \*, +, ?](https://rafalab.github.io/dsbook/string-processing.html#quantifiers-1){target="\_blank"}.

\*\*Key points\*\*

\* ```str\_replace()``` replaces the first instance of the detected pattern with a specified string.

\* Spaces are characters and R does not ignore them. Spaces are specified by the special character ```\\s```.

\* Additional quantifiers include ```\*```, ```+``` and ```?```. ```\*``` means 0 or more instances of the previous character. ```?``` means 0 or 1 instances. ```+``` means 1 or more instances.

\* Before removing characters from strings with functions like ```str\_replace()``` and ```str\_replace\_all()```, consider whether that replacement would have unintended effects.

\*Code\*

```{r, eval=FALSE, echo=TRUE}

# number of entries matching our desired pattern

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

sum(str\_detect(problems, pattern))

# inspect examples of entries with problems

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

str\_subset(problems, "inches")

str\_subset(problems, "''")

# replace or remove feet/inches words before matching

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

problems %>%

str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

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

str\_detect(pattern) %>%

sum()

# R does not ignore whitespace

identical("Hi", "Hi ")

# \\s represents whitespace

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

str\_subset(problems, pattern\_2)

# \* means 0 or more instances of a character

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

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

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

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

# test how \*, ? and + differ

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

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

nore\_or\_once = str\_detect(yes, "A1?B"),

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

# update pattern by adding optional spaces before and after feet symbol

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

problems %>%

str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

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

str\_detect(pattern) %>%

sum()

```

## Groups with Regex

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#groups){target="\_blank"}.

\*\*Key Points\*\*

\* Groups are defined using parentheses.

\* Once we define groups, we can use the function ```str\_match()``` to extract the values these groups define. ```str\_extract()``` extracts only strings that match a pattern, not the values defined by groups.

\* You can refer to the ith group with ```\\i```. For example, refer to the value in the second group with ```\\2```.

\*Code\*

```{r}

# define regex with and without groups

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

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

# create examples

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

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

s <- c(yes, no)

# demonstrate the effect of groups

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

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

# demonstrate difference between str\_match and str\_extract

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

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

# improve the pattern to recognize more events

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")

# final pattern

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

# combine stringr commands with the pipe

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

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

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

```

## Testing and Improving

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#testing-and-improving){target="\_blank"}.

\*\*Key points\*\*

\* Wrangling with regular expressions is often an iterative process of testing the approach, looking for problematic entries, and improving the patterns.

\* Use the pipe to connect \*\*stringr\*\* functions.

\* It may not be worth writing code to correct every unique problem in the data, but string processing techniques are flexible enough for most needs.

\*Code\*

```{r}

# function to detect entries with problems

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

}

# identify entries with problems

problems <- reported\_heights %>%

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

.$height

length(problems)

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

# find proportion of entries that fit the pattern after reformatting

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

index <- str\_detect(converted, pattern)

mean(index)

converted[!index] # show problems

```

## Assessment - String Processing Part 2

1. In the video, we use the function ```not\_inches``` to identify heights that were incorrectly entered

```{r, eval=FALSE, echo=TRUE}

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

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

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

ind

}

```

In this function, what TWO types of values are identified as not being correctly formatted in inches?

- [ ] A. Values that specifically contain apostrophes (‘), periods (.) or quotations (“).

- [X] B. Values that result in NA’s when converted to numeric

- [X] C. Values less than 50 inches or greater than 84 inches

- [ ] D. Values that are stored as a character class, because most are already classed as numeric.

2. Which of the following arguments, when passed to the function ```not\_inches```, would return the vector ```c(FALSE)```?

- [ ] A. ```c(175)```

- [ ] B. ```c(“5’8\””)```

- [X] C. ```c(70)```

- [ ] D. ```c(85)``` (the height of Shaquille O'Neal in inches)

3. Our function ```not\_inches``` returns the object ```ind```. Which answer correctly describes ```ind```?

- [X] A. ```ind``` is a logical vector of TRUE and FALSE, equal in length to the vector ```x``` (in the arguments list). TRUE indicates that a height entry is incorrectly formatted.

- [ ] B. ```ind``` is a logical vector of TRUE and FALSE, equal in length to the vector ```x``` (in the arguments list). TRUE indicates that a height entry is correctly formatted.

- [ ] C. ```ind``` is a data frame like our ```reported\_heights``` table but with an extra column of TRUE or FALSE. TRUE indicates that a height entry is incorrectly formatted.

- [ ] D. ```ind``` is a numeric vector equal to ```reported\_heights$heights``` but with incorrectly formatted heights replaced with NAs.

4. Given the following code

```{r}

s <- c("70", "5 ft", "4'11", "", ".", "Six feet")

s

```

What ```pattern``` vector yields the following result?

```{r, eval=FALSE, echo=TRUE}

str\_view\_all(s, pattern)

70

5 ft

4’11

.

Six feet

```

```{r, eval=FALSE, echo=TRUE}

pattern <- "\\d|ft"

str\_view\_all(s, pattern)

```

- [X] A. ```pattern <- "\\d|ft"```

- [ ] B. ```pattern <- "\d|ft"```

- [ ] C. ```pattern <- "\\d\\d|ft"```

- [ ] D. ```pattern <- "\\d|feet"```

5. You enter the following set of commands into your R console. What is your printed result?

```{r}

animals <- c("cat", "puppy", "Moose", "MONKEY")

pattern <- "[a-z]"

str\_detect(animals, pattern)

```

- [ ] A. TRUE

- [ ] B. TRUE TRUE TRUE TRUE

- [X] C. TRUE TRUE TRUE FALSE

- [ ] D. TRUE TRUE FALSE FALSE

6. You enter the following set of commands into your R console. What is your printed result?

```{r}

animals <- c("cat", "puppy", "Moose", "MONKEY")

pattern <- "[A-Z]$"

str\_detect(animals, pattern)

```

- [ ] A. FALSE FALSE FALSE FALSE

- [ ] B. FALSE FALSE TRUE TRUE

- [X] C. FALSE FALSE FALSE TRUE

- [ ] D. TRUE TRUE TRUE FALSE

7. You enter the following set of commands into your R console. What is your printed result?

```{r}

animals <- c("cat", "puppy", "Moose", "MONKEY")

pattern <- "[a-z]{4,5}"

str\_detect(animals, pattern)

```

- [X] A. FALSE TRUE TRUE FALSE

- [ ] B. TRUE TRUE FALSE FALSE

- [ ] C. FALSE FALSE FALSE TRUE

- [ ] D. TRUE TRUE TRUE FALSE

8. Given the following code

```{r, eval=FALSE, echo=TRUE}

animals <- c(“moose”, “monkey”, “meerkat”, “mountain lion”)

------------

str\_detect(animals, pattern)

```

Which TWO “pattern” vectors would yield the following result?

```{r, eval=FALSE, echo=TRUE}

[1] TRUE TRUE TRUE TRUE

```

```{r}

animals <- c("moose", "monkey", "meerkat", "mountain lion")

pattern <- "mo\*"

str\_detect(animals, pattern)

```

```{r}

animals <- c("moose", "monkey", "meerkat", "mountain lion")

pattern <- "mo?"

str\_detect(animals, pattern)

```

- [X] A. ```pattern <- "mo\*"```

- [X] B. ```pattern <- "mo?"```

- [ ] C. ```pattern <- "mo+"```

- [ ] D. ```pattern <- "moo\*"```

9. You are working on some data from different universities. You have the following vector

```{r}

schools <- c("U. Kentucky", "Univ New Hampshire", "Univ. of Massachusetts", "University Georgia", "U California", "California State University")

schools

```

You want to clean this data to match the full names of each university

```{r, eval=FALSE, echo=TRUE}

> final

[1] "University of Kentucky" "University of New Hampshire" "University of Massachusetts" "University of Georgia"

[5] "University of California" "California State University"

```

What of the following commands could accomplish this?

```{r}

schools %>%

str\_replace("^Univ\\.?\\s|^U\\.?\\s", "University ") %>%

str\_replace("^University of |^University ", "University of ")

```

- [ ] A.

```{r, eval=FALSE, echo=TRUE}

schools %>%

str\_replace("Univ\\.?|U\\.?", "University ") %>%

str\_replace("^University of |^University ", "University of ")

```

- [X] B.

```{r, eval=FALSE, echo=TRUE}

schools %>%

str\_replace("^Univ\\.?\\s|^U\\.?\\s", "University ") %>%

str\_replace("^University of |^University ", "University of ")

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

schools %>%

str\_replace("^Univ\\.\\s|^U\\.\\s", "University") %>%

str\_replace("^University of |^University ", "University of ")

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

schools %>%

str\_replace("^Univ\\.?\\s|^U\\.?\\s", "University") %>%

str\_replace("University ", "University of ")

```

10. Rather than using the ```pattern\_with\_groups``` vector, you accidentally write in the following code

```{r}

problems <- c("5.3", "5,5", "6 1", "5 .11", "5, 12")

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

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

```

What is your result?

- [X] A. ```[1] "5'3" "5'5" "6 1" "5 .11" "5, 12"```

- [ ] B. ```[1] “5.3” “5,5” “6 1” “5 .11” “5, 12”```

- [ ] C. ```[1] “5’3” “5’5” “6’1” “5 .11” “5, 12”```

- [ ] D. ```[1] “5’3” “5’5” “6’1” “5’11” “5’12”```

11. You notice your mistake and correct your pattern regex to the following

```{r}

problems <- c("5.3", "5,5", "6 1", "5 .11", "5, 12")

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

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

```

What is your result?

- [ ] A. ```[1] “5’3” “5’5” “6 1” “5 .11” “5, 12”```

- [ ] B. ```[1] “5.3” “5,5” “6 1” “5 .11” “5, 12”```

- [X] C. ```[1] "5'3" "5'5" "6'1" "5 .11" "5, 12"```

- [ ] D. ```[1] “5’3” “5’5” “6’1” “5’11” “5’12”```

12. In our example, we use the following code to detect height entries that do not match our pattern of x’y”.

```{r}

converted <- problems %>%

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

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

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

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

index <- str\_detect(converted, pattern)

converted[!index]

```

Which answer best describes the differences between the regex string we use as an argument in ```str\_replace("^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$", "\\1'\\2")``` and the regex string in ```pattern <- "^[4-7]\\s\*'\\s\*\\d{1,2}$"```?

- [ ] A. The regex used in ```str\_replace()``` looks for either a comma, period or space between the feet and inches digits, while the pattern regex just looks for an apostrophe; the regex in str\_replace allows for one or more digits to be entered as inches, while the pattern regex only allows for one or two digits.

- [ ] B. The regex used in ```str\_replace()``` allows for additional spaces between the feet and inches digits, but the pattern regex does not.

- [ ] C. The regex used in ```str\_replace()``` looks for either a comma, period or space between the feet and inches digits, while the pattern regex just looks for an apostrophe; the regex in str\_replace allows none or more digits to be entered as inches, while the pattern regex only allows for the number 1 or 2 to be used.

- [X] D. The regex used in ```str\_replace()``` looks for either a comma, period or space between the feet and inches digits, while the pattern regex just looks for an apostrophe; the regex in str\_replace allows for none or more digits to be entered as inches, while the pattern regex only allows for one or two digits.

13. You notice a few entries that are not being properly converted using your ```str\_replace``` and ```str\_detect``` code

```{r, eval=FALSE, echo=TRUE}

yes <- c("5 feet 7inches", "5 7")

no <- c("5ft 9 inches", "5 ft 9 inches")

s <- c(yes, no)

converted <- s %>%

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

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

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

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

str\_detect(converted, pattern)

[1] TRUE TRUE FALSE FALSE

```

It seems like the problem may be due to spaces around the words feet|foot|ft and inches|in. What is another way you could fix this problem?

```{r}

yes <- c("5 feet 7inches", "5 7")

no <- c("5ft 9 inches", "5 ft 9 inches")

s <- c(yes, no)

converted <- s %>%

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

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

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

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

str\_detect(converted, pattern)

```

- [X] A.

```{r, eval=FALSE, echo=TRUE}

converted <- s %>%

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

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

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

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

converted <- s %>%

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

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

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

```

- [ ] C.

```{r, eval=FALSE, echo=TRUE}

converted <- s %>%

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

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

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

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

converted <- s %>%

str\_replace\_all(“\\s”, “”) %>%

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

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

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

```

## Separate with Regex

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#the-extract-function){target="\_blank"}.

\*\*Key Point\*\*

\* The ```extract()``` function behaves similarly to the separate() function but allows extraction of groups from regular expressions.

\*Code\*

```{r}

# first example - normally formatted heights

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

tab <- data.frame(x = s)

# the separate and extract functions behave similarly

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

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

# second example - some heights with unusual formats

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

tab <- data.frame(x = s)

# separate fails because it leaves in extra characters, but extract keeps only the digits because of regex groups

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

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

```

## Using Groups and Quantifiers

The textbook for this section is available [here; through 24.10](https://rafalab.github.io/dsbook/string-processing.html#testing-and-improving){target="\_blank"}.

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 entires 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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

s <- "Hi "

cat(s)

identical(s, "Hi")

```

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

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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:

```{r, eval=FALSE, echo=TRUE}

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.

```{r}

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

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

}

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")

}

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:

```{r}

new\_heights %>%

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

select(original, height) %>%

arrange(height) %>%

View() # Open XQuartz-app to run this command

```

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

```{r}

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

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#string-splitting){target="\_blank"}.

\*\*Key Points\*\*

\* The function ```str\_split()``` splits a string into a character vector on a delimiter (such as a comma, space or underscore). By default, ```str\_split()``` generates a list with one element for each original string. Use the function argument ```simplify=TRUE``` to have ```str\_split()``` return a matrix instead.

\* The ```map()``` function from the \*\*purrr\*\* package applies the same function to each element of a list. To extract the ith entry of each element x, use ```map(x, i)```.

\* ```map()``` always returns a list. Use ```map\_chr()``` to return a character vector and ```map\_int()``` to return an integer.

\*Code\*

```{r}

# read raw murders data line by line

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

lines <- readLines(filename)

lines %>% head()

# split at commas with str\_split function, remove row of column names

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

x %>% head()

col\_names <- x[[1]]

x <- x[-1]

# extract first element of each list entry

if(!require(purrr)) install.packages("purrr")

library(purrr)

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

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

# extract columns 1-5 as characters, then convert to proper format - NOTE: DIFFERENT FROM VIDEO

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

# more efficient code for the same thing

dat <- x %>%

transpose() %>%

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

setNames(col\_names) %>%

as.data.frame()

# the simplify argument makes str\_split return a matrix instead of a list

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

col\_names <- x[1,]

x <- x[-1,]

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

setNames(col\_names) %>%

mutate\_all(parse\_guess)

```

## Case Study - Extracting a Table from a PDF

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#case-study-3-extracting-tables-from-a-pdf){target="\_blank"}.

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

```{r}

data("research\_funding\_rates")

research\_funding\_rates

```

The data come from a [paper](https://www.pnas.org/content/112/40/12349.abstract){target="\_blank"} 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:

```{r}

if(!require(pdftools)) install.packages("pdftools")

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 ```txt``` 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:

```{r}

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

```

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

```{r}

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

```

\*\*Looking at the download\*\*

Examining this object,

```{r}

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

```{r}

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:

```{r}

tab <- tab[[1]]

```

By examining this object,

```{r}

tab %>% head

```

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

```{r}

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:

```{r}

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:

```{r}

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:

```{r}

the\_names\_2

```

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

```{r}

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:

```{r}

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:

```{r}

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:

```{r}

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

```

## Recoding

The textbook for this section is available [here](https://rafalab.github.io/dsbook/string-processing.html#recode){target="\_blank"}.

\*\*Key points\*\*

\* Change long factor names with the ```recode()``` function from the \*\*tidyverse\*\*.

\* Other similar functions include ```recode\_factor()``` and ```fct\_recoder()``` in the \*\*forcats\*\* package in the \*\*tidyverse\*\*. The same result could be obtained using the ```case\_when()``` function, but ```recode()``` is more efficient to write.

\*Code\*

```{r}

# life expectancy time series for Caribbean countries

gapminder %>%

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

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

geom\_line()

# display long country names

gapminder %>%

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

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

distinct(country)

# recode long country names and remake plot

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' = "Trinidad")) %>%

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

geom\_line()

```

## Assessment - String Processing Part 3

Want even more practice with regular expressions? Complete the lessons and exercises in the [RegexOne](https://regexone.com){target="\_blank"} online interactive tutorial!

1.

```{r}

s <- c("5'10", "6'1\"", "5'8inches", "5'7.5")

tab <- data.frame(x = s)

```

If you use the extract code from our video, the decimal point is dropped. What modification of the code would allow you to put the decimals in a third column called “decimal”?

```{r}

extract(data = tab, col = x, into = c("feet", "inches", "decimal"),

regex = "(\\d)'(\\d{1,2})(\\.\\d+)?")

```

- [ ] A.

```{r,eval=FALSE, echo=TRUE}

extract(data = tab, col = x, into = c(“feet”, “inches”, “decimal”), regex = "(\\d)'(\\d{1,2})(\\.)?"

```

- [ ] B.

```{r,eval=FALSE, echo=TRUE}

extract(data = tab, col = x, into = c("feet", "inches", "decimal"), regex = "(\\d)'(\\d{1,2})(\\.\\d+)"

```

- [ ] C.

```{r,eval=FALSE, echo=TRUE}

extract(data = tab, col = x, into = c("feet", "inches", "decimal"), regex = "(\\d)'(\\d{1,2})\\.\\d+?"

```

- [X] D.

```{r,eval=FALSE, echo=TRUE}

extract(data = tab, col = x, into = c("feet", "inches", "decimal"), regex = "(\\d)'(\\d{1,2})(\\.\\d+)?")

```

2. You have the following table, ```schedule```

```{r,eval=FALSE, echo=TRUE}

>schedule

day staff

Monday Mandy, Chris and Laura

Tuesday Steve, Ruth and Frank

```

You want to turn this into a more useful data frame.

Which two commands would properly split the text in the “staff” column into each individual name? Select ALL that apply.

- [ ] A. ```str\_split(schedule$staff, ",|and")```

- [X] B. ```str\_split(schedule$staff, ", | and ")```

- [X] C. ```str\_split(schedule$staff, ",\\s|\\sand\\s")```

- [ ] D. ```str\_split(schedule$staff, "\\s?(,|and)\\s?")```

3. You have the following table, ```schedule```

```{r,eval=FALSE, echo=TRUE}

> schedule

day staff

Monday Mandy, Chris and Laura

Tuesday Steve, Ruth and Frank

```

What code would successfully turn your “Schedule” table into the following tidy table

```{r,eval=FALSE, echo=TRUE}

< tidy

day staff

<chr> <chr>

Monday Mandy

Monday Chris

Monday Laura

Tuesday Steve

Tuesday Ruth

Tuesday Frank

```

- [X] A.

```{r,eval=FALSE, echo=TRUE}

tidy <- schedule %>%

mutate(staff = str\_split(staff, ", | and ")) %>%

unnest()

```

- [ ] B.

```{r,eval=FALSE, echo=TRUE}

tidy <- separate(schedule, staff, into = c("s1","s2","s3"), sep = “,”) %>%

gather(key = s, value = staff, s1:s3)

```

- [ ] C.

```{r,eval=FALSE, echo=TRUE}

tidy <- schedule %>%

mutate(staff = str\_split(staff, ", | and ", simplify = TRUE)) %>%

unnest()

```

4. Using the gapminder data, you want to recode countries longer than 12 letters in the region “Middle Africa” to their abbreviations in a new column, “country\_short”. Which code would accomplish this?

```{r}

dat <- gapminder %>% filter(region == "Middle Africa") %>%

mutate(country\_short = recode(country,

"Central African Republic" = "CAR",

"Congo, Dem. Rep." = "DRC",

"Equatorial Guinea" = "Eq. Guinea"))

dat

```

- [ ] A.

```{r,eval=FALSE, echo=TRUE}

dat <- gapminder %>% filter(region == "Middle Africa") %>%

mutate(recode(country,

"Central African Republic" = "CAR",

"Congo, Dem. Rep." = "DRC",

"Equatorial Guinea" = "Eq. Guinea"))

```

- [ ] B.

```{r,eval=FALSE, echo=TRUE}

dat <- gapminder %>% filter(region == "Middle Africa") %>%

mutate(country\_short = recode(country,

c("Central African Republic", "Congo, Dem. Rep.", "Equatorial Guinea"),

c("CAR", "DRC", "Eq. Guinea")))

```

- [ ] C.

```{r,eval=FALSE, echo=TRUE}

dat <- gapminder %>% filter(region == "Middle Africa") %>%

mutate(country = recode(country,

"Central African Republic" = "CAR",

"Congo, Dem. Rep." = "DRC",

"Equatorial Guinea" = "Eq. Guinea"))

```

- [X] D.

```{r,eval=FALSE, echo=TRUE}

dat <- gapminder %>% filter(region == "Middle Africa") %>%

mutate(country\_short = recode(country,

"Central African Republic" = "CAR",

"Congo, Dem. Rep." = "DRC",

"Equatorial Guinea" = "Eq. Guinea"))

```

5. Import raw Brexit referendum polling data from Wikipedia:

```{r}

if(!require(stringr)) install.packages("stringr")

library(stringr)

url <- "https://en.wikipedia.org/w/index.php?title=Opinion\_polling\_for\_the\_United\_Kingdom\_European\_Union\_membership\_referendum&oldid=896735054"

tab <- read\_html(url) %>% html\_nodes("table")

polls <- tab[[5]] %>% html\_table(fill = TRUE)

```

You will use a variety of string processing techniques learned in this section to reformat these data.

Some rows in this table do not contain polls. You can identify these by the lack of the percent sign (%) in the Remain column.

Update ```polls``` by changing the column names to ```c("dates", "remain", "leave", "undecided", "lead", "samplesize", "pollster", "poll\_type", "notes")``` and only keeping rows that have a percent sign (%) in the remain column.

How many rows remain in the ```polls``` data frame?

```{r}

names(polls) <- c("dates", "remain", "leave", "undecided", "lead", "samplesize", "pollster", "poll\_type", "notes")

polls <- polls[str\_detect(polls$remain, "%"), -9]

nrow(polls)

```

6. The ```remain``` and ```leave``` columns are both given in the format "48.1%": percentages out of 100% with a percent symbol.

Which of these commands converts the ```remain``` vector to a proportion between 0 and 1?

Check all correct answers.

- [ ] A. ```as.numeric(str\_remove(polls$remain, "%"))```

- [ ] B. ```as.numeric(polls$remain)/100```

- [ ] C. ```parse\_number(polls$remain)```

- [ ] D. ```str\_remove(polls$remain, "%")/100```

- [X] E. ```as.numeric(str\_replace(polls$remain, "%", ""))/100```

- [X] F. ```parse\_number(polls$remain)/100```

7. The ```undecided``` column has some "N/A" values. These "N/A"s are only present when the ```remain``` and ```leave``` columns total 100%, so they should actually be zeros.

Use a function from \*\*stringr\*\* to convert "N/A" in the ```undecided``` column to 0. The format of your command should be ```function\_name(polls$undecided, "arg1", "arg2")```.

What function replaces ```function\_name```? ```str\_replace```

What argument replaces ```arg1```? ```N/A```

What argument replaces ```arg2```? ```0```

8. The ```dates``` column contains the range of dates over which the poll was conducted. The format is "8-10 Jan" where the poll had a start date of 2016-01-08 and end date of 2016-01-10. Some polls go across month boundaries (16 May-12 June).

The end date of the poll will always be one or two digits, followed by a space, followed by the month as one or more letters (either capital or lowercase). In these data, all month abbreviations or names have 3, 4 or 5 letters.

Write a regular expression to extract the end day and month from ```dates```. Insert it into the skeleton code below:

```{r, eval=FALSE, echo=TRUE}

temp <- str\_extract\_all(polls$dates, \_\_\_\_\_)

end\_date <- sapply(temp, function(x) x[length(x)]) # take last element (handles polls that cross month boundaries)

```

Which of the following regular expressions correctly extracts the end day and month when inserted into the blank in the code above? Check all correct answers.

- [ ] A. ```"\\d?\\s[a-zA-Z]?"```

- [X] B. ```"\\d+\\s[a-zA-Z]+"```

- [ ] C. ```"\\d+\\s[A-Z]+"```

- [X] D. ```"[0-9]+\\s[a-zA-Z]+"```

- [X] E. ```"\\d{1,2}\\s[a-zA-Z]+"```

- [ ] F. ```"\\d{1,2}[a-zA-Z]+"```

- [X] G. ```"\\d+\\s[a-zA-Z]{3,5}"```

# Section 4 Overview

In the \*\*Dates, Times, and Text Mining\*\* section, you will learn how to deal with dates and times in R and also how to generate numerical summaries from text data.

After completing this section, you will be able to:

\* Handle \*\*dates\*\* and \*\*times\*\* in R.

\* Use the \*\*lubridate\*\* package to parse dates and times in different formats.

\* Generate \*\*numerical summaries from text data\*\* and apply data visualization and analysis techniques to those data.

## Dates and Times

The textbook for this section is available [here](https://rafalab.github.io/dsbook/parsing-dates-and-times.html){target="\_blank"}.

\*\*Key points\*\*

\* Dates are a separate data type in R.The \*\*tidyverse\*\* includes functionality for dealing with dates through the \*\*lubridate\*\* package.

\* Extract the year, month and day from a date object with the ```year()```, ```month()``` and ```day()``` functions.

\* Parsers convert strings into dates with the standard YYYY-MM-DD format (ISO 8601 format). Use the parser with the name corresponding to the string format of year, month and day (```ymd()```, ```ydm()```, ```myd()```, ```mdy()```, ```dmy()```, ```dym()```).

\* Get the current time with the ```Sys.time()``` function. Use the ```now()``` function instead to specify a time zone.

\* You can extract values from time objects with the ```hour()```, ```minute()``` and ```second()``` functions.

\* Parsers convert strings into times (for example, ```hms()```). Parsers can also create combined date-time objects (for example, ```mdy\_hms()```).

\*Code\*

```{r}

# inspect the startdate column of 2016 polls data, a Date type

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

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

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

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

# ggplot is aware of dates

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

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

geom\_line()

# lubridate: the tidyverse date package

if(!require(lubridate)) install.packages("lubridate")

library(lubridate)

# select some random dates from polls

set.seed(2)

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

dates

# extract month, day, year from date strings

data.frame(date = dates,

month = month(dates),

day = day(dates),

year = year(dates))

month(dates, label = TRUE) # extract month label

# ymd works on mixed date styles

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

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

ymd(x)

# different parsers extract year, month and day in different orders

x <- "09/01/02"

ymd(x)

mdy(x)

ydm(x)

myd(x)

dmy(x)

dym(x)

now() # current time in your time zone

now("GMT") # current time in GMT

now() %>% hour() # current hour

now() %>% minute() # current minute

now() %>% second() # current second

# parse time

x <- c("12:34:56")

hms(x)

#parse datetime

x <- "Nov/2/2012 12:34:56"

mdy\_hms(x)

```

## Text mining

The textbook for this section is available [here](https://rafalab.github.io/dsbook/text-mining.html){target="\_blank"}.

\*\*Key points\*\*

\* The \*\*tidytext\*\* package helps us convert free form text into a tidy table.

\* Use ```unnest\_tokens()``` to extract individual words and other meaningful chunks of text.

\* Sentiment analysis assigns emotions or a positive/negative score to tokens. You can extract sentiments using ```get\_sentiments()```. Common lexicons for sentiment analysis are "bing", "afinn", "nrc" and "loughran".

With the exception of 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, counter-terrorism 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\*\*

See my [GitHub-repository on Trump Tweets](https://github.com/1965Eric/trump-tweets.git){target="\_blank"}.

## Assessment Part 1 - Dates, Times, and Text Mining

```{r}

options(digits = 3) # 3 significant digits

```

1. Which of the following is the standard ISO 8601 format for dates?

- [ ] A. MM-DD-YY

- [X] B. YYYY-MM-DD

- [ ] C. YYYYMMDD

- [ ] D. YY-MM-DD

2. Which of the following commands could convert this string into the correct date format?

```{r, eval=FALSE, echo=TRUE}

dates <- c("09-01-02", "01-12-07", "02-03-04")

```

- [ ] A. ```ymd(dates)```

- [ ] B. ```mdy(dates)```

- [ ] C. ```dmy(dates)```

- [X] D. It is impossible to know which format is correct without additional information.

3. Load the ```brexit\_polls``` data frame from dslabs:

```{r}

data(brexit\_polls)

```

How many polls had a start date (startdate) in April (month number 4)?

```{r}

sum(month(brexit\_polls$startdate) == 4)

```

Use the ```round\_date()``` function on the ```enddate``` column with the argument ```unit="week"```. How many polls ended the week of 2016-06-12?

Read the documentation to learn more about ```round\_date()```.

```{r}

sum(round\_date(brexit\_polls$enddate, unit = "week") == "2016-06-12")

```

4. Use the ```weekdays()``` function from \*\*lubridate\*\* to determine the weekday on which each poll ended (```enddate```).

On which weekday did the greatest number of polls end?

```{r}

table(weekdays(brexit\_polls$enddate))

```

- [ ] A. Monday

- [ ] B. Tuesday

- [ ] C. Wednesday

- [ ] D. Thursday

- [ ] E. Friday

- [ ] F. Saturday

- [X] G. Sunday

```{r}

max(weekdays(brexit\_polls$enddate))

```

5. Load the ```movielens``` data frame from \*\*dslabs\*\*.

```{r}

data(movielens)

```

This data frame contains a set of about 100,000 movie reviews. The ```timestamp``` column contains the review date as the number of seconds since 1970-01-01 (epoch time).

Convert the ```timestamp``` column to dates using the \*\*lubridate\*\* as\_datetime() function.

Which year had the most movie reviews?

```{r}

dates <- as\_datetime(movielens$timestamp)

reviews\_by\_year <- table(year(dates))

names(which.max(reviews\_by\_year))

```

Which hour of the day had the most movie reviews?

```{r}

reviews\_by\_hour <- table(hour(dates))

names(which.max(reviews\_by\_hour))

```

## Assessment Part 2 - Dates, Times, and Text Mining

6. Project Gutenberg is a digital archive of public domain books. The R package \*\*gutenbergr\*\* facilitates the importation of these texts into R. We will combine this with the \*\*tidyverse\*\* and \*\*tidytext\*\* libraries to practice text mining.

Use these libraries and options:

```{r}

if(!require(gutenbergr)) install.packages("gutenbergr")

if(!require(tidytext)) install.packages("tidytext")

library(gutenbergr)

library(tidytext)

```

You can see the books and documents available in \*\*gutenbergr\*\* like this:

```{r}

gutenberg\_metadata

```

Use ```str\_detect()``` to find the ID of the novel \*Pride and Prejudice\*.

How many different ID numbers are returned?

```{r}

gutenberg\_metadata %>%

filter(str\_detect(title, "Pride and Prejudice"))

```

7. Notice that there are several versions of the book. The ```gutenberg\_works()``` function filters this table to remove replicates and include only English language works. Use this function to find the ID for \*Pride and Prejudice\*.

What is the correct ID number?

Read the ```gutenberg\_works()``` documentation to learn how to use the function.

```{r}

gutenberg\_works(title == "Pride and Prejudice")$gutenberg\_id

```

8. Use the ```gutenberg\_download()``` function to download the text for Pride and Prejudice. Use the \*\*tidytext\*\* package to create a tidy table with all the words in the text. Save this object as ```words``.

How many words are present in the book?

```{r}

book <- gutenberg\_download(1342)

words <- book %>%

unnest\_tokens(word, text)

nrow(words)

```

9. Remove stop words from the ```words``` object. Recall that stop words are defined in the ```stop\_words``` data frame from the \*\*tidytext\*\* package.

How many words remain?

```{r}

words <- words %>% anti\_join(stop\_words)

nrow(words)

```

10. After removing stop words, detect and then filter out any token that contains a digit from ```words``.

How many words remain?

```{r}

words <- words %>%

filter(!str\_detect(word, "\\d"))

nrow(words)

```

11. Analyze the most frequent words in the novel after removing stop words and tokens with digits.

How many words appear more than 100 times in the book?

```{r}

words %>%

count(word) %>%

filter(n > 100) %>%

nrow()

```

What is the most common word in the book?

```{r}

words %>%

count(word) %>%

top\_n(1, n) %>%

pull(word)

```

How many times does that most common word appear?

```{r}

words %>%

count(word) %>%

top\_n(1, n) %>%

pull(n)

```

12. Define the afinn lexicon.

```{r}

afinn <- get\_sentiments("afinn")

```

Note that this command will trigger a question in the R Console asking if you want to download the AFINN lexicon. Press 1 to select "Yes" (if using RStudio, enter this in the Console tab).

Use this ```afinn``` lexicon to assign sentiment values to ```words```. Keep only words that are present in both ```words``` and the ```afinn``` lexicon. Save this data frame as ```afinn\_sentiments```.

How many elements of words have sentiments in the afinn lexicon?

```{r}

afinn\_sentiments <- inner\_join(afinn, words)

nrow(afinn\_sentiments)

```

What proportion of words in afinn\_sentiments have a positive value?

```{r}

mean(afinn\_sentiments$value > 0)

```

How many elements of afinn\_sentiments have a value of 4?

```{r}

sum(afinn\_sentiments$value == 4)

```

## Final: Comprehensive Assessment

### Comprehensive Assessment: Puerto Rico Hurricane Mortality

### Project Introduction

On September 20, 2017, Hurricane Maria made landfall on Puerto Rico. It was the worst natural disaster on record in Puerto Rico and the deadliest Atlantic hurricane since 2004. However, Puerto Rico's official death statistics only tallied 64 deaths caused directly by the hurricane (due to structural collapse, debris, floods and drownings), an undercount that slowed disaster recovery funding. The majority of the deaths resulted from infrastructure damage that made it difficult to access resources like clean food, water, power, healthcare and communications in the months after the disaster, and although these deaths were due to effects of the hurricane, they were not initially counted.

In order to correct the misconception that few lives were lost in Hurricane Maria, statisticians analyzed how death rates in Puerto Rico changed after the hurricane and estimated the excess number of deaths likely caused by the storm. [This analysis](https://drive.google.com/file/d/16X9qtnPaD--2dPhpcwu7S53esafH59i9/preview){target="\_blank"} suggested that the actual number of deaths in Puerto Rico was 2,975 (95% CI: 2,658-3,290) over the 4 months following the hurricane, much higher than the original count.

We will use your new data wrangling skills to extract actual daily mortality data from Puerto Rico and investigate whether the Hurricane Maria had an immediate effect on daily mortality compared to unaffected days in September 2015-2017.

```{r}

options(digits = 3) # report 3 significant digits

```

### Puerto Rico Hurricane Mortality - Part 1

1. In the ```extdata``` directory of the \*\*dslabs\*\* package, you will find a PDF file containing daily mortality data for Puerto Rico from Jan 1, 2015 to May 31, 2018. You can find the file like this:

```{r}

fn <- system.file("extdata", "RD-Mortality-Report\_2015-18-180531.pdf", package="dslabs")

```

Find and open the file or open it directly from RStudio. On a Mac, you can type:

```{r}

system2("open", args = fn)

```

and on Windows, you can type:

```{r, eval=FALSE, echo=TRUE}

system("cmd.exe", input = paste("start", fn))

```

Which of the following best describes this file?

- [ ] A. It is a table. Extracting the data will be easy.

- [ ] B. It is a report written in prose. Extracting the data will be impossible.

- [X] C. It is a report combining graphs and tables. Extracting the data seems possible.

- [ ] D. It shows graphs of the data. Extracting the data will be difficult.

2. We are going to create a tidy dataset with each row representing one observation. The variables in this dataset will be year, month, day and deaths.

Use the \*\*pdftools\*\* package to read in ```fn``` using the ```pdf\_text``` function. Store the results in an object called txt.

```{r}

txt <- pdf\_text(fn)

class(txt)

str(txt)

length(txt)

```

Describe what you see in ```txt```.

- [ ] A. A table with the mortality data.

- [X] B. A character string of length 12. Each entry represents the text in each page. The mortality data is in there somewhere.

- [ ] C. A character string with one entry containing all the information in the PDF file.

- [ ] D. An html document.

3. Extract the ninth page of the PDF file from the object ```txt```, then use the ```str\_split``` function from the \*\*stringr\*\* package so that you have each line in a different entry. The new line character is ```\n```. Call this string vector ```x```.

Look at ```x```. What best describes what you see?

What kind of object is ```x```?

How many entries does ```x``` have?

```{r}

x <- str\_split(txt[9], "\n")

class(x)

length(x)

```

- [ ] A. It is an empty string.

- [ ] B. I can see the figure shown in page 1.

- [ ] C. It is a tidy table.

- [X] D. I can see the table! But there is a bunch of other stuff we need to get rid of.

4. Define ```s``` to be the first entry of the ```x``` object.

What kind of object is s?

How many entries does s have?

```{r}

s <- x[[1]]

class(s)

length(s)

```

5. When inspecting the string we obtained above, we see a common problem: white space before and after the other characters. Trimming is a common first step in string processing. These extra spaces will eventually make splitting the strings hard so we start by removing them.

We learned about the command ```str\_trim```that removes spaces at the start or end of the strings. Use this function to trim ```s``` and assign the result to ```s``` again.

After trimming, what single character is the last character of element 1 of ```s```?

```{r}

s <- str\_trim(s)

s[1] # print string, visually inspect last character

```

6. We want to extract the numbers from the strings stored in ```s```. However, there are a lot of non-numeric characters that will get in the way. We can remove these, but before doing this we want to preserve the string with the column header, which includes the month abbreviation.

Use the ```str\_which``` function to find the row with the header. Save this result to ```header\_index```.

Hint: find the first string that matches the pattern ```"2015"``` using the ```str\_which``` function.

What is the value of ```header\_index```?

```{r}

header\_index <- str\_which(s, pattern="2015")[1]

header\_index

```

7. We want to extract two objects from the header row: ```month``` will store the month and ```header``` will store the column names.

Save the content of the header row into an object called ```header```, then use ```str\_split``` to help define the two objects we need.

What is the value of month? Use ```header\_index``` to extract the row. The separator here is one or more spaces. Also, consider using the ```simplify``` argument.

What is the third value in header?

```{r}

tmp <- str\_split(s[header\_index], pattern="\\s+", simplify=TRUE)

month <- tmp[1]

header <- tmp[-1]

month

header[3]

```

### Puerto Rico Hurricane Mortality - Part 2

8. Notice that towards the end of the page defined by ```s``` you see a \*"Total"\* row followed by rows with other summary statistics. Create an object called ```tail\_index``` with the index of the \*"Total"\* entry.

What is the value of ```tail\_index```?

```{r}

tail\_index <- str\_which(s, "Total")

tail\_index

```

9. Because our PDF page includes graphs with numbers, some of our rows have just one number (from the y-axis of the plot). Use the ```str\_count``` function to create an object ```n``` with the count of numbers in each row.

How many rows have a single number in them? You can write a regex for a number like this ```\\d+```.

```{r}

n <- str\_count(s, "\\d+")

sum(n == 1)

```

10. We are now ready to remove entries from rows that we know we don't need. The entry ```header\_index``` and everything before it should be removed. Entries for which ```n``` is 1 should also be removed, and the entry ```tail\_index``` and everything that comes after it should be removed as well.

How many entries remain in ```s```?

```{r}

out <- c(1:header\_index, which(n==1), tail\_index:length(s))

s <- s[-out]

length(s)

```

11. Now we are ready to remove all text that is not a digit or space. Do this using regular expressions (regex) and the ```str\_remove\_all``` function.

In regex, using the ```^``` inside the square brackets ```[]``` means \*not\*, like the ```!``` means \*not\* in ```!=```. To define the regex pattern to catch all non-numbers, you can type ```[^\\d]```. But remember you also want to keep spaces.

Which of these commands produces the correct output?

- [ ] A.

```{r, eval=FALSE, echo=TRUE}

s <- str\_remove\_all(s, "[^\\d]")

```

- [ ] B.

```{r, eval=FALSE, echo=TRUE}

s <- str\_remove\_all(s, "[\\d\\s]")

```

- [X] C.

```{r, eval=FALSE, echo=TRUE}

s <- str\_remove\_all(s, "[^\\d\\s]")

```

- [ ] D.

```{r, eval=FALSE, echo=TRUE}

s <- str\_remove\_all(s, "[\\d]")

```

12. Use the ```str\_split\_fixed``` function to convert ```s``` into a data matrix with just the day and death count data:

```{r}

s <- str\_split\_fixed(s, "\\s+", n = 6)[,1:5]

```

Now you are almost ready to finish. Add column names to the matrix: the first column should be ```day``` and the next columns should be the ```header```. Convert all values to numeric. Also, add a column with the month. Call the resulting object ```tab```.

What was the mean number of deaths per day in September 2015?

```{r}

tab <- s %>%

as\_data\_frame() %>%

setNames(c("day", header)) %>%

mutate\_all(as.numeric)

mean(tab$"2015")

```

What is the mean number of deaths per day in September 2016?

```{r}

mean(tab$`2016`)

```

Hurricane Maria hit Puerto Rico on September 20, 2017. What was the mean number of deaths per day from September 1-19, 2017, before the hurricane hit?

```{r}

mean(tab$`2017`[1:19])

```

What was the mean number of deaths per day from September 20-30, 2017, after the hurricane hit?

```{r}

mean(tab$`2017`[20:30])

```

13. Finish it up by changing ```tab``` to a tidy format, starting from this code outline:

```{r, eval=FALSE, echo=TRUE}

tab <- tab %>% \_\_\_\_\_\_\_\_(year, deaths, -day) %>%

mutate(deaths = as.numeric(deaths))

tab

```

What code fills the blank to generate a data frame with columns named "day", "year" and "deaths"?

```{r}

tab <- tab %>% gather(year, deaths, -day) %>%

mutate(deaths = as.numeric(deaths))

tab

```

- [ ] A. separate

- [ ] B. unite

- [X] C. gather

- [ ] D. spread

14. Make a plot of deaths versus day with color to denote year. Exclude 2018 since we have no data. Add a vertical line at day 20, the day that Hurricane Maria hit in 2017.

```{r}

tab %>% filter(year < 2018) %>%

ggplot(aes(day, deaths, color = year)) +

geom\_line() +

geom\_vline(xintercept = 20) +

geom\_point()

```

Which of the following are TRUE?

- [X] A. September 2015 and 2016 deaths by day are roughly equal to each other.

- [ ] B. The day with the most deaths was the day of the hurricane: September 20, 2017.

- [X] C. After the hurricane in September 2017, there were over 100 deaths per day every day for the rest of the month.

- [X] D. No days before September 20, 2017 have over 100 deaths per day.